On the Efficiency of Several VM Provisioning Strategies for Workflows with Multi-threaded Tasks on Clouds

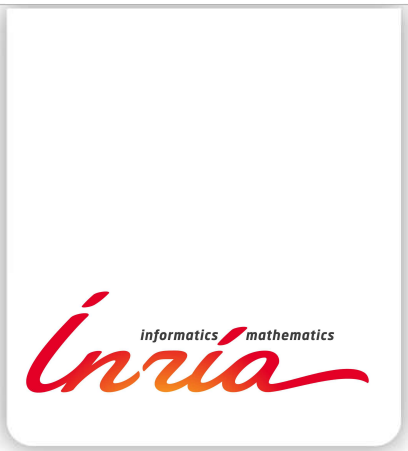
Marc E. Frincu, Stéphane Genaud, Julien Gossa

On the Efficiency of Several VM Provisioning Strategies for Workflows with Multi-threaded Tasks on Clouds

Marc E. Frincu , Stéphane Genaud , Julien Gossa

RESEARCH REPORT
N° 8449
January 2014
Project-Team AlGorille

ISSN 0249-6399 ISRN INRIA/RR--8449--FR+ENG



On the Efficiency of Several VM Provisioning Strategies for Workflows with Multi-threaded Tasks on Clouds

Marc E. Frincu *, Stéphane Genaud *, Julien Gossa *

Project-Team AlGorille

Research Report n° 8449 — January 2014 — 30 pages

Abstract: Cloud computing promises the delivery of on-demand pay-per-use access to unlimited resources. Using these resources requires more than a simple access to them as most clients have certain constraints in terms of cost and time that need to be fulfilled. Therefore certain scheduling heuristics have been devised to optimize the placement of client tasks on allocated virtual machines. The applications can be roughly divided in two categories: independent bag-of-tasks and workflows. In this paper we focus ourselves on the latter and investigate a less studied problem, i.e., the effect the virtual machine allocation policy has on the scheduling outcome. For this we look at how workflow structure, execution time, virtual machine instance type affect the efficiency of the provisioning method when cost and makespan are considered.
To aid our study we devised a mathematical model for cost and makespan in case single or multiple instance types are used. While the model allows us to determine the boundaries for two of our extreme methods, the complexity of workflow applications requires a more experimental approach to determine the general relation. For this we considered simulations of real application workflows and synthetic ones, covering most of the possible cases.
Results have shown the need for probabilistic selection methods in case small and heterogeneous execution times are used, while for large homogeneous ones the best algorithm is clearly noticed. Several other conclusions regarding the efficiency of powerful instance types as compared to weaker ones, and of dynamic methods against static ones are also made.

Key-words: workflow scheduling, virtual machine provisioning, cloud computing, cost and makespan modeling

* Icube, CNRS - Université de Strasbourg, France.

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Efficacité comparée de plusieurs stratégies de provisionnement de VM pour des workflows de tâches multi-threadées sur Cloud

Résumé : Le cloud computing ouvre la perspective de ressources illimitées sur la base d'un paiement à l'utilisation. L'utilisation de ces ressources exige cependant plus que les primitives de base de gestion des machines virtuelles proposées à l'utilisateur, car elles ne prennent pas en compte les contraintes de coût et de temps d'exécution. Par conséquent, des heuristiques d'ordonnancement doivent être proposées pour optimiser le placement des tâches des clients sur les machines virtuelles allouées. Les applications peuvent être divisées en deux catégories principales : celles composées de tâches indépendantes et les workflows. Dans cet article nous nous focalisons sur les workflows, et nous nous concentrons sur le problème assez peu étudié de l'effet de la politique d'allocation des machines virtuelles sur l'ordonnancement. Pour cela, nous examinons comment la structure, le temps d'exécution, et le type d'instance de machine virtuelle influencent l'efficacité de la méthode d'allocation et d'ordonnancement.

L'étude s'appuye sur un modèle mathématique modélisant le coût et le temps de fin dans le cas ou on utilise un seul type d'instance, ou différent types d'instance de machines virtuelles. Bien que le modèle permette de déterminer des bornes pour deux des méthodes extrêmes, la complexité des workflows nécessite une approche expérimentale basée sur des cas variés pour déterminer une relation générale entre les paramètres.

Les résultats montrent la nécessité de méthodes probabilistes de sélection dans le cas où les temps d'exécution des tâches sont courts et hétérogènes, tandis que pour des temps d'exécution homogènes le meilleur algorithme est clairement identifié. Plusieurs autres conclusions sont également tirées concernant l'efficacité de l'utilisation de types d'instance plus ou moins puissantes, ou de l'emploi de méthodes statiques ou au contraire dynamiques.

Mots-clés : ordonnancement de workflows, provisionnement de machines virtuelles, cloud computing, modélisation de coût et durée

1 Introduction

Cloud computing has received an increasing attention in the past years due to its promise of delivering on-demand pay-per-use access to virtually unlimited resources. This aspect has also brought a lot of challenges in terms of standards and technical solutions for addressing security, scaling, interoperability, brokering or management issues. Problems such as interoperability and brokering are becoming the central focus of many research initiatives due to the increasing number of new providers which leads to a wide array of heterogeneous offers from which clients must select the ones best for their goals. This selection problem can be solved manually or automatically and while the former could work for small short running applications, elastic applications that are complex enough not to facilitate manual selection necessitate a smarter software driven selection. This automatic selection needs to properly allocate cloud resources –in this paper we only deal with Virtual Machines (VMs)– in order to meet some client objectives.

The scientific community deals with two kinds of applications, those without any dependencies, i.e., bag-of-tasks (BoTs), and those in which the execution of some tasks depend on the successful completion of previous ones, i.e., workflows. For the latter case we have execution paths which can be deterministic –the path can be determined a priori, which is usually the case of directed acyclic graphs (DAGs)– or non-deterministic –the execution path is usually determined at runtime and consists of loop, split and join constructs [6]. Commercial cloud applications can be divided similarly in single service oriented –e.g., weather, stock exchange services– or workflow oriented –e.g., bank transfers, online reservations, social websites.

One of the problems linked to VM selection is to decide when to allocate them and which tasks to assign to them. In a virtualized environment such as a cloud the issue is three folded as it requires scheduling on three levels: (1) deciding when a new VM is required; (2) finding the appropriate Physical Machine (PM) for it– a problem related to bin packing [3]; and (3) scheduling the tasks on the VMs depending on various user given objectives. Based on the provider policies it can influence one or more of these stages. In this paper we focus on the first and third steps which are usually controllable by clients.

Many grid Scheduling Algorithms (SAs) for BoT have been extended and adapted to clouds [15, 16, 23, 30]. These papers show the various impact VM provisioning policies have on the same task scheduling method and outline their importance.

The problem of workflow scheduling has mostly tackled the aspect of extending previous grid SAs to rent cloud resources whenever needed and ignored, to our best knowledge (cf. Sect. 2), to study the impact of the VM provisioning on the scheduling policy. The focus of this paper is therefore to investigate this problem in the context of CPU intensive workflows where data communication does not play a major role. We show that the SA outcome is linked to the workflow structure and the provisioning method and the used VM instance type. This work extends the one previously presented in [12] where we restricted the study to only a special case of four existing workflows. In addition we focus our study on multi-threaded tasks since many cloud providers offer many multi-core VM instance types.

Given the above we argue that the primary contribution of this paper is two folded:

- to provide several VM allocation methods which we model and analyze in terms of cost and makespan. This allows for a better understanding of how they behave in ideal scenarios such as those with only parallel or sequential tasks. Cost and makespan are investigated and modeled in two cases, with and without VM boot/shutdown times and for multiple VM instance types (e.g., Table 1 depicts the pricing list for on-demand VM types offered by Amazon). To our knowledge this is the first attempt at creating such a model;

- to show that objectives such as cost and makespan are influenced not only by the workflow structure and task scheduling but by the VM provisioning strategy and VM instance types as well. This proves that there is more behind VM allocation than a simple "rent whenever needed the least expensive instance type". For this we present results for four classic workflow structures and also extend our analysis to randomly generated DAGs. This correlation between the previously mentioned properties could greatly improve the efficiency of resource management systems by allowing them to dynamically switch the allocation methods at runtime.

The rest of the paper is structured as follows: Section 2 depicts some of the main results in the field of cloud SAs for BoTs and workflows. Section 3 depicts the main VM provisioning policies and the three methods used to order tasks inside the workflow. It also presents some theoretical aspects related to the relation between the methods when one or more VM instance types are used as well as a study on their optimality (cf. Sect. 3.2). The experimental setup

Table 1: Amazon EC2 standard on-demand instance types and prices on Feb 26th 2013.

region	small	medium	large	xlarge	transfer out
US East Virginia	0.06	0.12	0.24	0.48	0.12
US West Oregon	0.06	0.12	0.24	0.48	0.12
US West California	0.065	0.13	0.26	0.52	0.12
EU Dublin	0.065	0.13	0.26	0.52	0.12
Asia Singapore	0.08	0.16	0.32	0.64	0.19
Asia Tokyo	0.088	0.175	0.350	0.700	0.201
Asia Sydney	0.08	0.16	0.32	0.64	0.19
SA Sao Paolo	0.08	0.16	0.32	0.64	0.25

is presented in Sect. 4. The case of real workflows is discussed in Sect. 4.1 while while the synthetically generated ones are addressed in Sect. 4.2. The conclusions and future work are addressed in Sect. 5.

2 Related Work on Cloud Scheduling

As a successor of Grid computing, Cloud computing inherited many of its problems, including those related to scheduling. Furthermore it also brought new ones, like scaling (or provisioning) which is strongly tied to the scheduling problem. Much work has been done to adapt existing grid SAs for clouds. These include both BoT and workflow oriented. Concerning the problem complexity we know the scheduling problem to be $\mathcal{NP} - complete$ and the *bin packing problem* of fitting multiple VMs on a PM to be $\mathcal{NP} - hard$ [3].

Although many SAs for clouds have been proposed for BoTs with some recent work [16, 23, 30] even analyzing the impact of VM provisioning on the SA, none have addressed the importance of the provisioning policy for the case of workflows. For BoTs tests have shown a dependency between the two which impacts both the makespan gain and the paid cost.

One of the first works to show the cost of deploying scientific workflows on clouds is [8] where three cloud migration scenarios are depicted: (1) use clouds sporadically to enhance the local infrastructure, (2) deploy the entire application on the cloud but keep the data locally, and (3) deploy both data and application on the clouds. Focus is however not on the VM provisioning method itself but on the impact renting resources from clouds has on the cost.

Most commercial clouds leave the client to decide when to provision VMs. Their concern is primarily related to VM to PM assignment and use simple allocation methods based on Round Robin (Amazon EC2[1]), least connections and weighted least connections (Rackspace[2]). Other simple policies include Least-Load or Rotating-Scheduling [15].

As already mentioned the majority of the results concerning workflow SAs have ignored the impact VM provisioning has on the SA and focused on extending existing algorithms – e.g., HEFT [33], CPA [26] and Gain [27]. Results in this direction include SHEFT [18] –an extension of HEFT which uses cloud resources whenever needed to decrease the makespan below a deadline–, CPA versions for determining the needed number of VMs a workflow requires [6], or Gain versions for clouds [21].

A few papers have proposed novel methods [1, 5, 19, 21, 24] but show little interest in how VM provisioning impacts metrics such as makespan or cost. Other approaches include: auction based scheduling [24]; HCOC [1] which relies on the Path Clustering Heuristic (PCH) [2] and uses an approach similar to SHEFT for provisioning cloud resources; Particle Swarm Optimization [25, 31]; Genetic Algorithms [31]; and Ant Colony Optimization [31] methods.

A distinct category seems to be that of map-reduce workflows. These are highly parallel applications consisting of two phases, a map phase in which keys are generated based on map tasks and a reduce phase in which keys are read and results are produced. In [17] two algorithms for minimizing VM rent costs are proposed: List and First-Fit –sorts prices and the corresponding VMs are allocated to map and reduce tasks –and Deadline-aware Tasks Packing –uses the estimated deadline to schedule map tasks.

Most of the presented solutions offer VM allocation policies tailored to the specific SA and while we do not question their efficiency we underline the fact that when grid SAs are extended for clouds [1, 6, 18] a greater importance should be given to both the VM allocation policy and the used instance types –e.g., those from Amazon EC2.

[1] http://aws.amazon.com/ec2/ (accessed Dec 7th 2012)
[2] http://www.rackspace.com/blog/announcing-cloud-load-balancing-private-beta/ (accessed Dec 7th 2012)

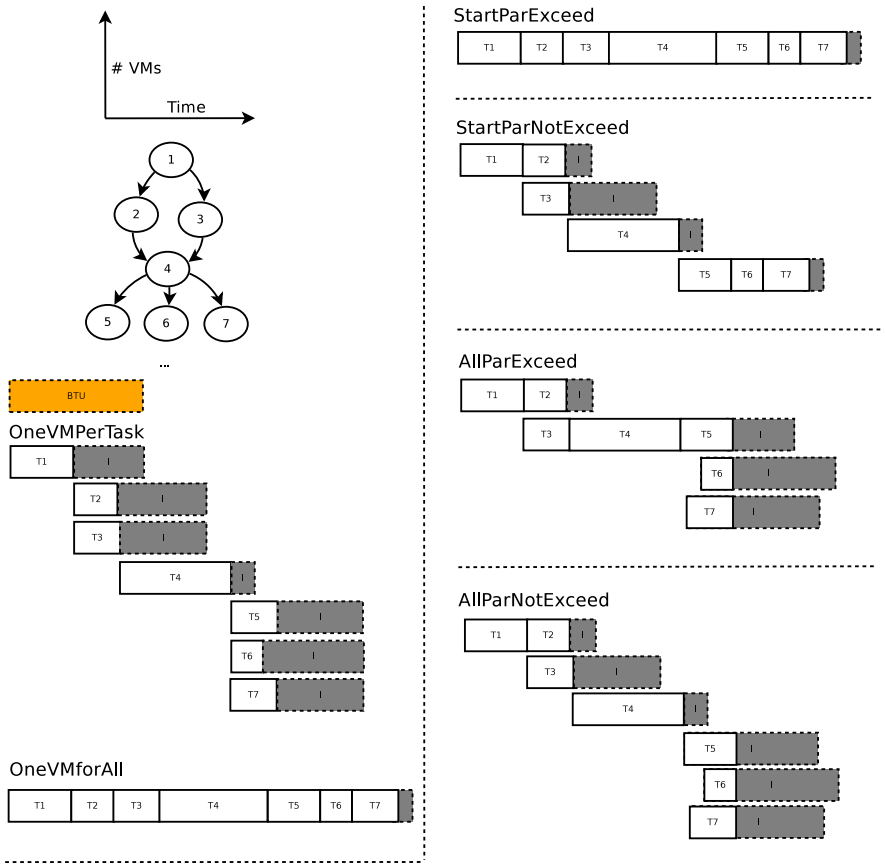


Figure 1: Basic VM provisioning policies.

A comparison between several workflow types and a hybrid private cloud + public cloud SA called HCOC is given in [1]. However the provisioning strategy apparently follows only a one VM for each task approach.

A special category of SA is that which relies on exact optimal solutions such as those provided by linear programming techniques. Again the majority of papers deal with BoTs [4, 20, 29, 32] while only a few address the case of workflows [13]. We argue that these solutions are more suited for static environments or for comparing the efficiency of other policies designed for online scenarios. Arguments include the time needed to compute the solution when large numbers of tasks and resources are used as well as their requirement to have a complete view of the world which in online scenarios is not possible.

3 VM Provisioning

VM provisioning refers to *how* VMs are allocated: reusing existing (idle) ones; renting a new one for new tasks or when the task execution time exceeds the remaining Billing Time Unit (BTU) [23]; considering or not boot time; etc. The BTU represents the logical unit used by providers to count for how long a VM has been rented. In general it is equal to 3,600s but recently Google introduced for its Compute Engine a billing per minute which takes effect after the first ten minutes of usage which are billed as a whole [14]. For our purposes we assume a uniform BTU for all our VMs.

The notion of provisioning is also used as a loosely synonym for scaling as it allows the VM resource pool to elastically scale based on demand. In [23] it has already been shown that for BoTs the VM provisioning affects objectives such as *idle time*, *cost* and *makespan* of the schedule. In this work we investigate workflows and propose as comparison six basic (cf. Fig. 1) and two dynamic (cf. Fig. 2) methods for VM provisioning. We further assume that all six basic methods use only homogeneous VM instances –e.g., small instances from Amazon EC2.

Figures 1 and 2 exemplifies them on a simple CSTEM sub-workflow [10] consisting of one initial task and six subsequent tasks. The unused BTU time is indicated by the dark *I*-marked rectangles while the default length of a BTU is represented by the rectangle marked *BTU*.

OneVMforAll assigns a single VM to execute all possible tasks. In order to achieve this,

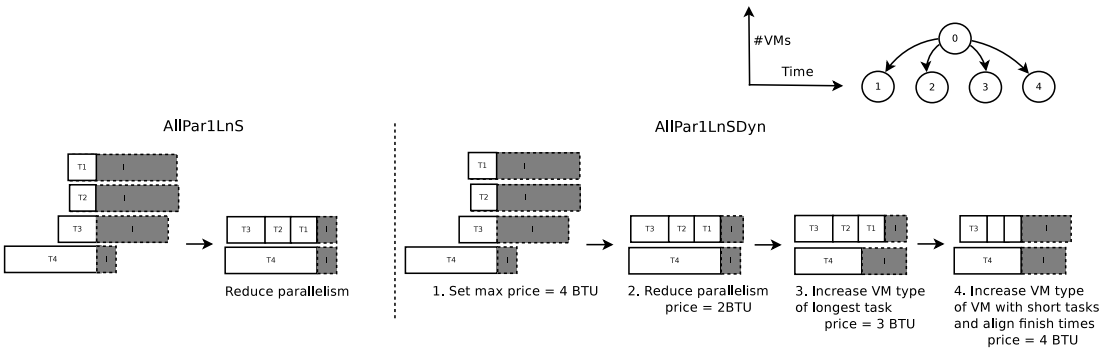


Figure 2: Dynamic VM provisioning policies.

workflow tasks need to be ranked[33]. This method is a extremely efficient in reducing cost irrespective of the workflow structure (cf. Sect. 3.2) but by serializing parallel tasks it produces long makespans.

OneVMperTask does exactly the opposite by assigning a new VM to each task. Thus it achieves an optimal makespan (cf. Sect. 3.2) but at the expense of maximizing the rent cost.

The rest of the proposed allocation methods are intermediaries trying to cover cases in which workflows have mixed sequential and parallel structures. The two *StartPar** methods are designed for workflows in which the maximum number of parallel tasks is given by the initial tasks while the *AllPar** methods target general workflows exhibiting an arbitrary degree of parallelism.

StartParExceed assigns a new VM to every initial workflow task. The rest of the tasks are scheduled sequentially on the initially rented VMs –for a given task the VM with the largest execution time is chosen. If a single initial task exists this heuristics becomes equivalent to *OneVMforAll*.

StartParNotExceed is similar to the previous one but tasks whose execution times exceed the remaining BTU free time are assigned to new VMs.

AllParNotExceed assigns each parallel task to its own VM –existing or new. New VMs are added when the number of parallel tasks exceeds the number of VMs or if a task execution time exceeds the assigned VM's remaining BTU time. For each task the algorithm tries to find the best fit. To achieve this it sorts both VM remaining BTU time and task execution times ascending. Another optimization is that it starts each new VM so that all parallel tasks finish at the same time –maximizing the remaining BTU time for each newly rented VM.

AllParExceed is similar to the previous but exceeding the remaining BTU does not lead to new VMs being rent.

It can be noticed that each provisioning strategy provides a different result in terms of allocated VMs, cost and makespan. The *OneVMperTask* and *OneVMforAll* policies represent upper limits with regard to the cost respectively makespan.

StartParNotExceed usually allocates more VMs than *StartParExceed* and allows in certain scenarios for the number of tasks executed in parallel to exceed the maximum value determined by the number of initial tasks.

The *AllParExceed* strategy fully exploits task parallelism. It reduces makespan by running independent tasks in parallel and also costs as sequential tasks are allocated on the same VM. *AllParNotExceed* is similar but the resulted number of rented VMs is larger. The efficiency of the two is however limited if little or no parallelism exists in the workflow.

Overall the strategies that tend to allocate more VMs are better suited for tasks with large data dependencies where the VM should be as close as possible to the data. On the other hand these give large idle times resulting in a waste of budget. The way in which these provisioning policies impact various workflow types is discussed in Sect. 4.

Finally, the two dynamic VM provisioning methods start from an initial provisioning achieved through either *AllParExceed* or *AllParNotExceed* and attempt to further reduce cost and makespan by taking advantage of task parallelism and faster VM instance types. Figure 2 presents them for a workflow having four parallel tasks.

AllPar1LnS tries to decrease task parallelism by executing in sequence multiple short parallel tasks whose total lengths are about the same as longest tasks. Each set of sequential short tasks is mapped onto a single VM, while the long tasks are still scheduled in parallel to different VMs. The reduction is performed only after tasks are ranked, inside each level, by execution time. A

more complex approach has been proposed in [21].

While *AllPar1LnS* reduces costs alone, a further optimized version called *AllPar1LnSDyn* tries to decrease the makespan inside each level by attempting to increase the VM speed within a given level budget:

First the parallelism is reduced as in *AllPar1LnS*. Then the worst budget for the level is computed based on the value given by a provisioning method using *OneVMperTask*. The algorithm will attempt to reduce the execution time of the longest task –which is always scheduled separately –by assigning the next fastest VM. If succeeded it checks to see whether the makespan is still determined by that task or it shifted to another VM (cf. Fig. 2). If the former case is true it continues to increase the speed of the VM within the budget until the makespan shifts to another VM or all VM types have been used. In case the makespan shifts to another VM, the SA tries to reduce it below the execution time of the longest task by increasing –within the budget– the speed of the VM. If this fails either due to exceeding the budget or because the makespan is still larger, the increase in speed is rolled back to the last valid configuration, i.e., one in which the budget is not exceeded and the level makespan is dictated by the longest task.

As an example consider the prices in Table 1 for the case of the Amazon cloud. Given a BTU of 500s and tasks 1–4 in Fig. 2 and assuming their execution times to be of: 100s, 120s, 130s, and 400s, for the parallelism reduction phase (*AllPar1LnS*) we get two VMs, the first one running the first three tasks in sequence (total execution time of 350s) and the second one the longest one. Given a single region, e.g., Virginia, the cost for using the cheapest two VMs is \$0.12. Considering *AllPar1LnSDyn* we compute the worst cost: \$0.24. We now attempt to reduce makespan by increasing the VM type executing the longest task. Assuming a speed-up of 1.6 (cf. Sect. 3.2) and a price of \$0.12 for the medium instance we have the new execution time is only 250s and our total cost is \$0.18. As we require our makespan to be given by the longest task we also need to reduce the execution times for our three tasks on the first VM. Since we are still below the worst price we attempt to increase the VM instance for them too. By doing so we obtain a total time of 218.75s $<$ 250s. As the total amount of cost reached the worst case price (\$0.24) we cannot further attempt to minimize the makespan. However, we were able of reducing the makespan by 150s while paying as much as in the worst case of *OneVMperTask*.

The proposed policies cover most of the approaches taken in literature so far. *OneVMperTask* is the usual approach when using cloud resources when the private cluster becomes full (e.g., SHEFT). In addition, algorithms like Gain or CPA require that only the VM assigned to a particular task to be augmented. Also, it is suited for parallel tasks where in order to minimize makespan each one must be executed on a separate VM. Optimizations of this simple allocation have been proposed in [21] and they include among others, parallelism reduction which in our case is incorporated in *AllPar1LnS*. While this method only reduces cost we have proposed our own enhancement in the form of *AllPar1LnSDyn* which given a cost upper bound –provided by *OneVMperTask* – tries to also reduce makespan by augmenting the VMs initially assigned using *AllPar1LnS*. In the context of these optimized algorithms the *{Start|All}Par[Not]Exceed* methods represent solutions which attempt to make full use of parallelism while not renting additional VMs in case sequential tasks need to be scheduled as well. Finally *OneVMforAll* is a method best suited for sequential workflows where rent costs are minimized in case a single VM is used throughout the execution.

3.1 Task ordering

Given a static scenario, DAG tasks need to be ranked so that their execution takes place in the right order. Three main methods exist: rank based, level based and cluster based. In this work we focus our attention on the first two methods. The first one lies at the foundation of the HEFT algorithm. For each task a rank based on execution and transfer times is computed. Tasks are then ordered descending by their ranks and scheduled accordingly.

Level based scheduling is a similar ranking strategy that orders tasks based on their level in the workflow [21]. Each level is made up of parallel tasks. The difference from HEFT is that it allows greater flexibility in deciding the order of scheduling inside a level –e.g., randomly, or by execution time. Its knowledge on which tasks can be executed independently and in parallel can lead to improvements where parallelism is reduced [21]. It can also be combined with the CPA [6] algorithm.

Cluster based scheduling comprises methods that cluster tasks located on the same path in order to reduce communication costs. Inside each cluster tasks are then ordered based on a ranking algorithm. Examples include PCH and HCOC.

Considering the eight VM provisioning methods, we have used level based ranking together with *AllPar[Not]Exceed*, *AllPar1LnS* and *AllPar1LnSDyn* while the HEFT priority ranking has been applied to the rest (each method is prefixed with *H-* in the experiments depicted in Sect. 4).

3.2 Theoretical Considerations Regarding Cost and Makespan

OneVMperTask and *OneVMforAll* are extreme provisioning methods providing the limits on the number of VMs allocated for a given number of parallel or sequential workflow tasks (cf. Propositions 1 and 2). These two types of tasks represent extreme cases that can be found in workflows. For instance MapReduce[3] has at least one layer of tasks that execute simultaneously. The number of used VMs is maximized in this way. On the other hand we have sequential workflows in which each task has to wait for its predecessor to finish execution before continuing. For this kind of workflows a single VM would be more cost effective than one for each task. As shown next optimal VM allocation solutions can be derived for makespan or cost for these two extreme workflows. As the majority of workflows are a combination of these two patterns we argue and prove that they represent extreme cases that give the boundaries in terms of cost and makespan.

We use *OneVMperTask* and *OneVMforAll* as landmarks in establishing the relation between the proposed basic methods and leave outside of our study the dynamic methods as they represent optimized versions of *AllPar[Not]Exceed* and their efficiency is highly dependent on the lengths of the parallel tasks' execution times.

We also provide some theoretical results when multiple VM instance types are considered. Two cases with and without boot/shutdown times are analyzed. Our focus is mainly on the cost and makespan ordering and boundaries and on how they relate when a faster and a slower VM are compared. As it will be seen in Sect. 3.2.2 whether the price is higher, lower or similar depends on a clear relation between speed-up, cost increase and BTU size.

To take advantage of the fact that most providers use multi-core VMs we assume all tasks are parallelizable and multi-threaded, a fairly realistic assumption when considering scientific applications, e.g., mathematical workflows relying on MPI. This allows each task to use all available cores during execution and also to simplify our model by allowing at most one task per VM to execute at all times. However it should be noted that in case of single-threaded tasks the costs for using a larger VM instance type remain unchanged for parallel tasks while they are n times higher for sequential ones (n represents the cost increase). The reason is that for most cloud providers we pay the number of used cores –e.g., a VM with two cores is twice as expensive as one with a single core.

Let e_i be a sequence of positive numbers with $i = \overline{1,n}$ representing the tasks execution times and BTU a fixed positive integer denoting the time unit used to rent a resource. We assume that a task's execution begins from the moment it starts transferring input data and ends when it has finished writing its output. This simplified model is suited for CPU intensive tasks that spend most of the time performing the actual processing.

One final notation convention is that for simplicity, unless otherwise stated, we use $\sum$ as equivalent to $\sum_{i=1}^{n}$. Proofs for the various propositions and remarks in Sects. 3.2.1 and 3.2.2 are given in the appendix.

Table 2 contains the list of notations used from this point onward.

3.2.1 Single instance type

A simple scenario without considering any of the boot/shutdown times is assumed in this case. It allows us to see the overall relations between the various methods in an ideal case free of any noise induced by the specific characteristics of different hypervisors [22].

We first define the cost and makespan for the two cases of *OneVMperTask* and *OneVMforAll* (cf. Definition 1) and proceed to prove their optimality for various special cases as noted by the following propositions and remarks.

Definition 1. *Given a single VM instance type the cost and makespan for the extreme cases of sequential and parallel execution of tasks are:*

- *OneVMperTask:*

$$\frac{\sum e_i}{BTU} \leq = \underbrace{\sum \left\lceil \frac{e_i}{BTU} \right\rceil}_{=cost_p} < \frac{\sum e_i}{BTU} + n \tag{1}$$

[3] http://hadoop.apache.org/ (accessed Jun 24th 2013)

Table 2: Main notations.

notation	description
e_i	execution time of task i
α_k	speed-up for VM instance type k
$gamma_k$	cost increase for VM instance type k
τ_i	boot/shutdown times for VM instance i
$cost^k_{algorithm}$	the cost when using *algorithm* with VM instances of type k
$makespan^k_{algorithm}$	the schedule makespan when using *algorithm* with VM instances of type k
$cost^k_p$	the cost when using *OneVMperTask* with VM instances of type k
$makespan^k_p$	the schedule makespan when using *OneVMperTask* with VM instances of type k
$cost^k_s$	the cost when using *OneVMforAll* with VM instances of type k
$makespan^k_s$	the schedule makespan when using *OneVMforAll* with VM instances of type k

$$makespan_p = \begin{cases} \max e_i & , \textit{parallel tasks} \\ \sum e_i & , \textit{sequential tasks} \end{cases} \tag{2}$$

- *OneVMforAll:*

$$\frac{\sum e_i}{BTU} \leq \underbrace{\left\lceil \frac{\sum e_i}{BTU} \right\rceil}_{cost_s} < \frac{\sum e_i}{BTU} + 1 \tag{3}$$

$$makespan_s = \sum e_i \tag{4}$$

Proposition 1. *For any given VM instance type OneVMforAll is optimal in terms of cost for both parallel and sequential tasks.*

Proposition 2. *OneVMperTask allocates the maximum number of possible VMs independently on the used VM instance type.*

***Remark* 1.** *A consequence of Prop. 2 is that it achieves an optimal makespan for both parallel and sequential tasks.*

***Remark* 2.** *OneVMforAll is optimal in terms of makespan for sequential tasks.*

Considering a multi-objective comprising of *cost* $\oplus$ *makespan* and depending on the structure of the DAG, intermediary methods could prove more efficient. Figure 3 shows how the proposed basic and dynamic provisioning methods transform from one into another when certain conditions occur.

For instance *AllParNotExceed* becomes equivalent to *AllParExceed* when all tasks ready to execute (i.e., tasks whose predecessors have completed executing) can fit any of the available VMs without exceeding the remaining BTU time. On the other hand if all ready tasks exceed the remaining BTU time it will allocate VMs the same way *OneVMperTask* does. Finally if there are no parallel tasks that share the same predecessor the method becomes identical to *StartParNotExceed*.

The transformation graph of *StartParNotExceed* is similar to that of the *AllParNotExceed* so we do not detail it here. Instead we focus on that of *AllParExceed*. This method becomes equivalent to *StartParExceed* if there is no common predecessor for any ready tasks. In turn the latter transforms into *OneVMforAll* if a single initial task exists.

The transformation of the two dynamic methods is simpler, with the *AllPar1LnSDyn* being the most general one. In case no speed-up is possible within the allocated budget it degrades into *AllPar1LnS*. This also degrades into either *AllParNotExceed* or *AllParExceed* (depending on the implementation) if –given several parallel tasks– there is no large enough task to accommodate the sequential execution of several shorter tasks.

Next, we are interested in relations between inflicted costs and produced makespans. Propositions 3, 4, 5 and 6 provide these orderings.

Proposition 3. *Given a set of parallel tasks we have the following cost ordering:*

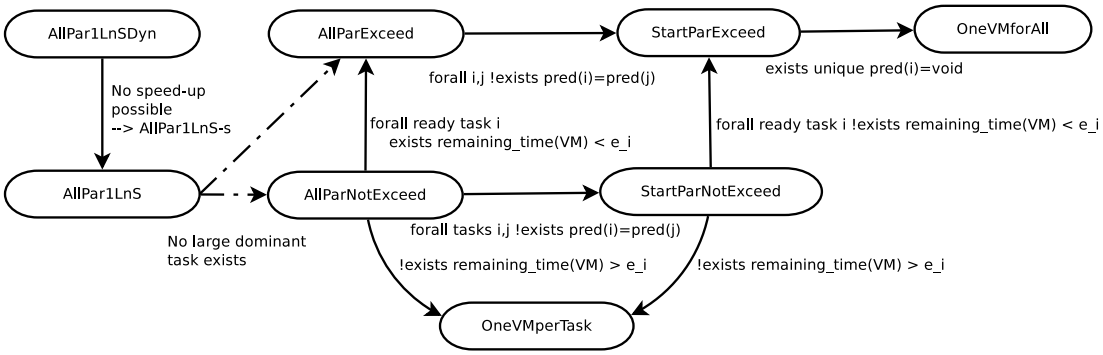


Figure 3: Relation diagram between various provisioning strategies.

$$\begin{aligned} cost_p &= cost_{AllParNotExceed} \\ &= cost_{AllParExceed} \\ &= cost_{StartParExceed} \\ &= cost_{StartParNotExceed} \\ &\geq cost_s \end{aligned}$$

Proposition 4. *Given a set of sequential tasks we have the following cost ordering:*

$$\begin{aligned} cost_p &\geq cost_{AllParNotExceed} \\ &= cost_{StartParNotExceed} \\ &\geq cost_{AllParExceed} \\ &= cost_{StartParExceed} \\ &= cost_s \end{aligned}$$

Proposition 5. *Given a set of parallel tasks we have the following makespan ordering:*

$$\begin{aligned} makespan_p &= makespan_{AllParExceed} \\ &= makespan_{AllParNotExceed} \\ &\leq makespan_{StartParNotExceed} \\ &\leq makespan_{StartParExceed} \\ &\leq makespan_s. \end{aligned}$$

Proposition 6. *Given a set of sequential tasks the makespan produced by each method is identical to that of OneVMperTask.*

From these propositions we can notice that the orderings of cost and makespan depends on the structure of the tasks. Given that many workflows are neither purely parallel nor sequential finding the right cost/makespan balance could depend on a combination between their structure and the deployed provisioning method.

For the single instance type, we are able to provide some general relations between arbitrary sequences (cf. Propositions 7 and 8).

Proposition 7. *Given a set of arbitrary tasks we have the following cost ordering:*

$$\begin{aligned} cost_p &\geq cost_{AllParNotExceed} \\ &\geq cost_{AllParExceed} \\ &\geq cost_{StartParExceed} \\ &\geq cost_s \end{aligned}$$

and

$$\begin{aligned} cost_p &\geq cost_{StartParNotExceed} \\ &\geq cost_{StartParExceed} \\ &\geq cost_s \end{aligned}$$

***Remark* 3.** *Regarding Proposition 7, the relation between $cost_{AllParExceed}$ and $cost_{StartParNotExceed}$, depends on e_i and the workflow structure.*

Considering one initial task followed by three parallel tasks i, j, k connected to it. If $e_i > BTU - e_{initial}$ we get $cost_{AllParExceed} = \lceil cost_{initial} + e_i \rceil + \lceil e_j \rceil + \lceil e_k \rceil$ and $cost_{StartParNotExceed} = \lceil cost_{initial} \rceil + \lceil e_i \rceil + \lceil e_j \rceil + \lceil e_k \rceil$, from which $cost_{AllParExceed} < cost_{StartParNotExceed}$. If $e_i \leq BTU - e_{initial}$ the two costs are identical. In addition when considering the workflow depicted in Fig. 1 we can clearly see that $cost_{AllParExceed} > cost_{StartParNotExceed}$. The reason is that *StartParNotExceed* schedules parallel tasks in sequence on the same VM as long as the BTU is not exceeded, which costs less than renting a VM for each one (cf. Proposition 3).

Proposition 8. *Given a set of arbitrary tasks we have the following makespan ordering:*

$$\begin{aligned} makespan_p &= makespan_{AllParNotExceed} \\ &= makespan_{AllParExceed} \\ &\leq makespan_{StartParNotExceed} \\ &\leq makespan_{StartParExceed} \\ &\leq makespan_s. \end{aligned}$$

In this section we have introduced some relations between costs and makespans when considering the proposed basic VM provisioning methods. These relations between our methods are difficult if not impossible to determine in case of multiple VM instances, due to the workflows' structure, the execution times, and the speed-up and cost of each type. Thus we focus next only on the extreme cases (as proven here) of *OneVMforAll* and *OneVMperTask*.

3.2.2 Multiple instance types

Cloud providers usually offer to clients more than one VM instance type. These can be modeled similarly as the single instance type case but require additional information such as their speed-up and cost increase. Hence we introduce two increasing monotonic sequences to account for the speed-up in execution time, α_m, and cost increase, γ_m. Given that clients usually pay the number of cores not the speed-up –which is not linear to the number of cores– we can generally assume $\gamma_k \geq \alpha_k$. The times needed to boot/shutdown a VM instance i of type k are modeled by a sequence τ_i^k. There is no general relation between the boot times of various instance types as they differ based on providers and VM characteristics [22]. Furthermore the boot time τ_i^k can also depend on the number of parallel booted VMs but for our purposes we assume a general case.

For the speed-up we notice two cases:

- tasks are inherently parallelizable and multi-threaded, i.e., they can take advantage of multi-core systems and run faster on them. In this case α_k represents the speed-up achieved by using the multi-core architecture;
- tasks are inherently serial, i.e., there is a single thread running for each task. In this case α_k depends on the speed of the VM processor and not on the number of cores.

As already mentioned in this paper we consider only the first case.

We have two possible scenarios: with and without boot/shutdown times. Both scenarios can be used in real life practical applications. The former is suited for offline or deadline based scheduling where we can anticipate the moment a task needs to start executing and we can pre-boot the VM to be ready at that moment. It also assumes that boot time is not part of the billing. The latter is applicable in cases where tasks arrive online without any knowledge on their start time. Hence VMs will be booted at the moment when a new task becomes ready for execution. In this case boot times are billed as part of the BTU. This billing model is used by many providers, e.g., Amazon EC2 or Rackspace, and means that clients pay resource not OS usage, i.e., the time that the core and RAM the VM instance is using and not the actual time the instance is usable. Other combinations are possible but we limit ourselves to those two.

The cost and makespan cf. Definition 1 are generalized as follows:

Definition 2. *Given multiple instance types and the boot/shutdown times for an arbitrary instance the cost and makespan for the extreme cases of are defined as follows:*

- *OneVMperTask:*

$$\frac{\sum \tau_i^k + \sum e_i}{\alpha_k BTU} \gamma_k \leq \underbrace{\sum \left\lceil \frac{\tau_i^k + e_i}{\alpha_k BTU} \right\rceil \gamma_k}_{cost_p^k} < \left(\frac{\sum \tau_i^k + \sum e_i}{\alpha_k BTU} + n \right) \gamma_k \tag{5}$$

$$makespan_p^k = \begin{cases} \max\left(\tau_i^k + \frac{e_i}{\alpha_k}\right) & , parallel\ tasks \\ \sum \tau_i^k + \frac{\sum e_i}{\alpha_k} & , sequential\ tasks \end{cases} \quad (6)$$

- *OneVMforAll:*

$$\frac{\tau_1^k + \sum e_i}{\alpha_k BTU}\gamma_k \leq \underbrace{\left\lceil \frac{\tau_1^k + \sum e_i}{\alpha_k BTU} \right\rceil \gamma_k}_{cost_s^k} < \left(\frac{\tau_1^k + \sum e_i}{\alpha_k BTU} + 1\right)\gamma_k \quad (7)$$

$$makespan_s^k = \tau_1^k + \frac{\sum e_i}{\alpha_k} \quad (8)$$

Without boot/shutdown times. We set $\tau_i^k = 0, \forall i, k$ and prove that based on α_k and γ_k there are three cases regarding the relation between costs. In addition some undefined boundaries can be determined if the BTU size falls within certain limits (cf. Proposition 9).

Proposition 9. *The following conditions (as depicted in Fig. 4) can be placed on the order of the lower/upper bounds of the cost intervals:*

- *if* $\sum e_i \leq BTU$ *and* $n \geq 2$*:* $cost_s^k < cost_s^{k+1} < cost_p^k < cost_p^{k+1}$
- *else* ($\frac{\alpha_k \gamma_{k+1}}{\alpha_{k+1}\gamma_k} = f$)*:*
  - $\boxed{f < 1}$*:* $\inf cost_s^{k+1} = \inf cost_p^{k+1} < \inf cost_s^k = \inf cost_p^k < \sup cost_s^k < \sup cost_p^k$.

    In addition:

$$\begin{cases} \sup cost_p^{k+1} \leq \inf cost_s^k & , BTU \leq \frac{(1-f)\sum e_i}{nf\alpha_{k+1}} \\ \sup cost_p^{k+1} \geq \sup cost_p^k & , BTU \leq \frac{(f-1)\sum e_i}{n(\alpha_k - f\alpha_{k+1})} \\ \sup cost_s^{k+1} \geq \sup cost_p^k & , BTU \leq \frac{(f-1)\sum e_i}{n\alpha_k - f\alpha_{k+1}} \text{ and } n < \frac{\gamma_{k+1}}{\gamma_k} \end{cases}$$

  - $\boxed{f > 1}$*:* $\inf cost_s^k = \inf cost_p^k < \inf cost_s^{k+1} = \inf cost_p^{k+1}$ *and* $\sup cost_s^k < \sup cost_s^{k+1} < \sup cost_p^{k+1}$ *and* $\sup cost_s^k < \sup cost_p^k < \sup cost_p^{k+1}$.

    In addition:

$$\begin{cases} \sup cost_p^k \leq \sup cost_s^{k+1} & , BTU \leq \frac{(f-1)\sum e_i}{n\alpha_k - f\alpha_{k+1}} \\ \inf cost_p^{k+1} \geq \sup cost_p^k & , BTU \leq \frac{(f-1)\sum e_i}{n\alpha_k} \end{cases}$$

  - $\boxed{f = 1}$*:* $\inf cost_s^k = \inf cost_p^k = \inf cost_s^{k+1} = \inf cost_p^{k+1}$.

    In addition: $\sup cost_p^k = \sup cost_p^{k+1}$ *and* $\sup cost_s^k = \sup cost_s^{k+1}$ *if* $\alpha_k = \alpha_{k+1}$.

We now exemplify how different providers fit in these conditions. For this we consider two cases for the speed-up as indicated by the STATA benchmarks: $\alpha^1 = (1, 1.6, 2.1, 2.7)$ and $\alpha^2 = (1, 1.8, 2.8, 4.1)$, where α_k^i represents the speed-up corresponding to having 2^{k-1} cores on the VM.

For the cost increase we have $\gamma^1 = (1, 2, 4, 8)$ for the majority of the providers –i.e., Amazon EC2 (cf. Table 1), Google Cloud and HP Cloud. However clouds like Rackspace or IBM Cloud exhibit price increases larger than the #cores increase ($\gamma^2 = (1, 2, 4, 10)$), respectively smaller (e.g., for 64 bit custom installed Linux $\gamma^3 = (1, 1.22, 1.97, 3.91)$).

Figure 4 exemplifies the relations in Proposition 9. When $f = 1$ all lower bounds are equal and as the speed increases the upper bound increases too. It also states that we can safely determine these bounds. The equality means that we pay exactly the speed-up, e.g., we pay twice as much for double the speed. The only time this happens (within a ± 0.03 accuracy) is (1) when combining α_2^2 (1.8) and IBM's γ_2^3 (1.97), and (2) a special case of using 32 bit RedHat on IBM cloud ($\gamma_1^{3'} = 1$ and $\gamma_2^{3'} = 1.85$) with α_1^2 respectively α_2^2.

When $f < 1$ the lower bound of the cost for an improved instance type will be less or equal to the initial one. This indicates that there are cases in which using faster resources is

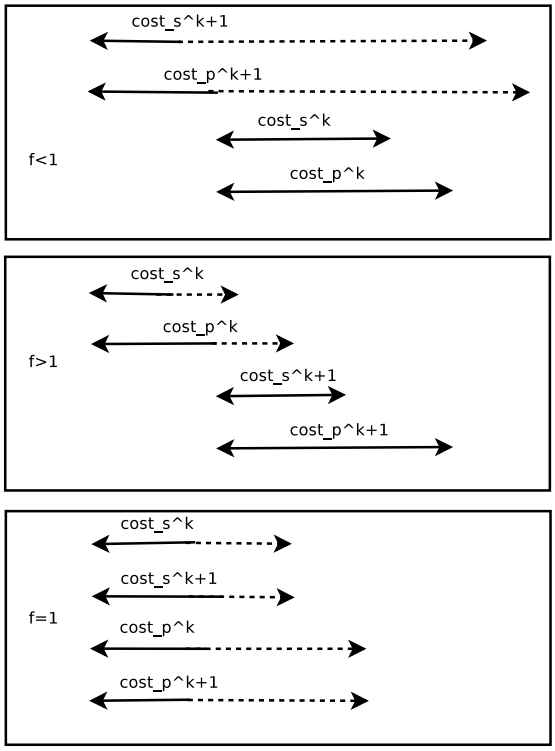


Figure 4: Representation of the three conditions listed in Proposition 9 for $\sum e_i > BTU$ (not on scale).

cheaper. The loose ends of the upper bound costs for the improved instance however also state that under certain conditions there is a good chance to pay more. This condition provides the broadest range for the cost of an augmented instance. Given our speed-up and cost vectors only the IBM cloud obeys this condition and only for α_1^1.

Finally the $f > 1$ condition reflects most pricing models used today. It effectively specifies that when using faster instance a client usually pay at least the same price as when using the slower instance.

Additional conditions put on the BTU length allow us to further determine the relations between the upper boundaries of the costs. For instance in case $f < 1$ or $f > 1$ we have a special condition on the BTU which makes the cost intervals for two consecutive instance types be disjoint. This indicates that the costs paid in the two cases will not overlap. These *exclusive* conditions are met when $BTU \leq \frac{(1-f)\sum e_i}{nf\alpha_{k+1}}$ respectively $BTU \leq \frac{(f-1)\sum e_i}{n\alpha_k}$. They allows us to safely assume that by increasing the instance type we will definitely pay less respectively more. It is noticed from the above mentioned inequalities that the BTU depends on variables such as the execution times and the number of tasks. When investigating the case with boot/shutdown times we will provide relations independent on these variables that depend only on τ_i^k and α_k which can be known in advance for online cases.

Another interesting relation between cost boundaries takes place when $f = 1$ and $\alpha_k = \alpha_{k+1}$. In this case the costs when using two different VM instance types are the same. It must be noticed that this happens only for tasks without multi-thread support for which running on multi-core architectures does not bring any advantages.

Regarding the relations between makespans we have the following ordering as given by Propositions 10, 11 and 12.

Proposition 10. *The makespan produced by each method for parallel tasks has the following ordering:*

$$\begin{cases} makespan_s^k < makespan_s^{k-1} < makespan_s^{k-2} \\ makespan_p^k < makespan_p^{k-1} < makespan_p^{k-2} \\ makespan_p^k \leq makespan_s^k < makespan_s^{k-1} \\ makespan_p^k < makespan_p^{k-1} \leq makespan_s^{k-1} \end{cases} \tag{9}$$

Proposition 11. *If in addition to Proposition 10 we have that* $\frac{\alpha_k}{\alpha_{k-1}} > \frac{\sum e_i}{\max e_i}$ *then the following ordering exists:* $makespan_p^k \leq makespan_s^k \leq makespan_p^{k-1} \leq makespan_s^{k-1}$

Proposition 12. *The makespan produced by each method for sequential tasks has the following ordering:* $makespan_s^k = makespan_p^k = \frac{\sum e_i}{\alpha_k} > makespan_s^{k+1} = makespan_p^{k+1}$.

Proposition 10 presents the makespan ordering when parallel tasks are considered. While the relations are clear regarding the advantage of faster instance types they leave open the issue of how *OneVMperTask* is related to *OneVMforAll* in case the latter uses a faster VM. Proposition 11 sheds some light in this aspect by linking the execution time to the speed-up. Hence, the faster *OneVMforAll* will be better then the slower *OneVMperTask* only if a long dominant task exists. This basically indicates that in this special case a serial execution of parallel tasks on faster VMs provides a better makespan than a parallel one on a slower VM. This could provide effective in terms of costs. Given for instance three tasks with the execution times being of 100s, 100s respectively 1,000s and a BTU=3600s we get that by executing them using *OneVMforAll* on the EC2 medium instance costs \$0.12 and produces a makespan of 750. Alternatively with *OneVMperTask* and EC2 small instances the cost is \$0.18 and the makespan is 1,000s. Furthermore using *OneVMperTask* with EC2 medium instances gives a cost of \$0.32 and a makespan of 625s. This translates in 16% makespan gain at a cost increase of 266% for *OneVMperTask* compared to *OneVMforAll* with medium instance types.

From these propositions, we notice that although for makespan we have a clear ordering for cost we can specify only boundaries based on the value for f and some relations between BTU and execution time. The reason is that while makespan is easily optimized based solely on the structure of the workflow, cost depends on many variables such as speed-up, price and workflow structure.

With boot/shutdown times. We consider $\tau_i^k > 0, \forall i,k$ to have arbitrary values that depend on the VM hypervisor.

Boot time can greatly influence the schedule and for methods similar in terms for makespan it tends to penalize the ones that rent more VMs. Tests comparing *OneVMperTask* with *AllParExceed* –two methods that produce optimal makespans– have shown that when boot/shutdown times are considered the former constantly gives longer makespans and that the difference is greater as workflows become more sequential. Given constant boot and shutdown times that are independent on the number of booted instances (as in the case of Amazon EC2) and Pareto shaped execution times (shape had a value of two and the scale was set to 500) we obtained that the makespan produced by *OneVMperTask* for MapReduce DAGs is greater by 2% than the one produced by *AllParExceed*. For sequential DAGs the difference goes up to 5%. Regarding costs there are no significant differences and where observed it usually implies the rent of at most one extra VM in case boot/shutdown times are considered. More generally, considering the small boot/shutdown times as compared to the BTU, we have that in the worst case, i.e., $BTU - \tau_i^k < e_i < BTU$ the cost is doubled by parallelism.

We provide in Proposition 13 the relations between costs when boot/shutdown times into consideration for the general case of multiple VM instance types. These remain the same as depicted in Figure 4 for the single instance type case.

Proposition 13. *The relations between costs given in Proposition 9 hold with the following modifications:*

– $\boxed{f<1}$*:* $\inf cost_s^{k+1} = \inf cost_p^{k+1} < \inf cost_s^k = \inf cost_p^k < \sup cost_s^k < \sup cost_p^k$ *if:*

$$\begin{cases} \sum t_i^k = \tau_1^k \\ f\tau_1^{k+1} - \tau_1^k < (1-f)\sum e_i \end{cases}$$

In addition:

$$\begin{cases} \sup cost_p^{k+1} \leq \inf cost_s^k, & BTU \leq \frac{(1-f)\sum e_i + \sum \tau_i^k - f\sum \tau_i^{k+1}}{nf\alpha_{k+1}} \\ or\, BTU \leq \frac{\sum \tau_i^k - \sum \tau_i^{k+1}}{n\alpha_{k+1}} & \\ \sup cost_p^{k+1} \geq \sup cost_p^k, & BTU \leq \frac{(f-1)\sum e_i + f\sum \tau_i^{k+1-\sum \tau_i^k}}{n(\alpha_k - f\alpha_{k+1})} \\ \sup cost_s^{k+1} \geq \sup cost_p^k, & BTU \leq \frac{(f-1)\sum e_i + f\tau_1^{k+1-\sum \tau_i^k}}{n\alpha_k - f\alpha_{k+1}} \\ and\ n < \frac{\gamma_{k+1}}{\gamma_k} & \end{cases}$$

– $\boxed{f > 1}$*:* $\inf cost_s^k = \inf cost_p^k < \inf cost_s^{k+1} = \inf cost_p^{k+1}$ *and* $\sup cost_s^k < \sup cost_s^{k+1} < \sup cost_p^{k+1}$ *and* $\sup cost_s^k < \sup cost_p^k < \sup cost_p^{k+1}$ *if:*

$$\begin{cases} f\tau_1^{k+1} - \tau_1^k \geq (1-f)\sum e_i \\ f\tau_1^{k+1} - \tau_1^k \geq (\alpha_k - f\alpha_{k+1})BTU + (1-f)\sum e_i \\ f\sum \tau_i^{k+1} - \sum \tau_i^k \geq (\alpha_k - f\alpha_{k+1})nBTU + (1-f)\sum e_i \end{cases}$$

In addition:

$$\begin{cases} \sup cost_p^k < \sup cost_s^{k+1} & , BTU \leq \frac{f\tau_1^{k+1} + (f-1)\sum e_i - \sum \tau_i^k}{n\alpha_k - f\alpha_{k+1}} \\ or\, BTU \leq \frac{\tau_1^{k+1} - \sum \tau_i^k}{n\alpha_k - \alpha_{k+1}} & \\ \inf cost_p^{k+1} \geq \sup cost_p^k & BTU \leq \frac{f\sum \tau_i^{k+1} + (f-1)\sum e_i - \sum \tau_i^k}{n\alpha_k} \\ or\, BTU \leq \frac{\sum \tau_i^{k+1} - \sum \tau_i^k}{n\alpha_k}, & \end{cases}$$

– $\boxed{f = 1}$*:* $\inf cost_s^k = \inf cost_p^k = \inf cost_s^{k+1} = \inf cost_p^{k+1}$ *if* $\sum \tau_i^{k+1} = \sum \tau_i^k$ *and* $\tau_1^{k+1} = \tau_1^k$.

In addition: $\sup cost_p^k = \sup cost_p^{k+1}$ *and* $\sup cost_s^k = \sup cost_s^{k+1}$ *if* $BTU = \frac{\alpha_k - \alpha_{k+1}}{n(\sum \tau_i^{k+1} - \sum \tau_i^k)}$.

***Remark* 4.** *It can be noticed that for* $\tau_i^k = 0, \forall i, k$ *the conditions are identical to those in Proposition 9.*

We already mentioned that the *exclusive* conditions when $f > 1$ and $f < 1$ can also be expressed based solely on the relation between BTU, speed-up and boot/shutdown time.

In general for these conditions to be true we have for $f > 1$ and $f < 1$ that $BTU \leq \frac{\sum \tau_i^{k+1} - \sum \tau_i^k}{n\alpha_k}$, respectively $BTU \leq \frac{\sum \tau_i^k - \sum \tau_i^{k+1}}{n\alpha_{k+1}}$. While for the second one is questionable whether or not there are VM types for which the boot time decreases as their capabilities increase, the first one is easier to achieve.

It has been shown in [22] that although for providers like Amazon the boot time is constant no matter the instance type or the number of parallel boots, others like Rackspace or Microsoft Azure, show an increase in the boot time as the instance type grows. Without losing generality we can consider $\tau_i^k = a$ and $\tau_i^{k+1} = ua, u \geq 1, a = ct.$. We obtain that $na(u-1) \geq n\alpha_k BTU$, which leads to:

$$BTU \leq \frac{a(u-1)}{\alpha_k} \tag{10}$$

It is thus not unrealistic to assume that if a provider would change the BTU to a per-second basis or some other value to fit Relation 10 the exclusive case could become a reality. Taking for instance Azure and considering $\tau_i^1 \approx 350$s and $\tau_i^2 \approx 360$s we obtain that $BTU < 16$s.

Regarding makespan when considering the boot/shutdown times the ordering depends on the properties of the τ_i^k sequence. Taking for instance the case of Amazon EC2 where boot times are constant and independent on the number of parallel booted VMs –e.g., $\tau_i^k = \tau_i^{k+1} = \tau$– we notice that Proposition 10 remains unchanged while Proposition 12 becomes:

Proposition 14. *The makespan produced by each method for sequential tasks has the following ordering:*

$$\begin{cases} makespan_s^k < makespan_s^{k-1} < makespan_s^{k-2} \\ makespan_p^k < makespan_p^{k-1} < makespan_p^{k-2} \\ makespan_s^k \leq makespan_s^{k-1} < makespan_p^{k-1} \\ makespan_s^k < makespan_p^k \leq makespan_p^{k-1} \end{cases} \tag{11}$$

Proposition 15. *In addition if* $\tau > \frac{\sum e_i}{n-1} \frac{\alpha_k - \alpha_{k-1}}{\alpha_{k-1}\alpha_k}$ *we have that* $makespan_s^k < makespan_s^{k-1} \leq makespan_p^k$.

These last two propositions show us how different the ordering for sequential tasks is when boot/shutdown times are considered as compared to ignoring them (cf. Proposition 12). The straightforward ordering becomes conditioned by the boot/shutdown time τ and presents a case in which a slower instance sequentializing tasks (*OneVMforAll*) on a single VM can be better than a faster one executing each task on its VM (*OneVMperTask*). The condition lies in the speed-up being greater than the boot/shutdown times (cf. Prop. 15).

3.3 Special Considerations for Single-Threaded Tasks

While in this paper we focus on multi-threaded tasks, we make a few remarks on the single-threaded case. To easily adapt the proposed model to this scenario we need to treat each core as an individual VM. In the case of *OneVMperTask* the sums (cf. Relations 5 and 6) will iterate from $\overline{1, \lceil \frac{n}{\#cores_k} \rceil}$, where $\#cores_k$ represents the number of cores instance type k has. Depending on the number of tasks we end up with at most $n \mod \#cores_k$ unused but rented VMs. Considering *OneVMforAll*, as it requires a single VM for all tasks this means that at most one core would be used and Relations 7 and 8 would remain unchanged except for the sum which will iterate over the same values as in the *OneVMperTask* case.

While this constraint is a limiting one it allows to link the two models and also for more general models to relate to the theoretical results obtained in this paper.

4 Experiments

While our model can be used to asses the behavior of our methods in extreme cases like pure parallel or sequential workflows, the wide array of workflow structures used by scientific applications requires the model to be reinforced with observations obtained from experimental results.

In order to verify how the provisioning method relates to the workflow structure and instance type we considered two experiments, one involving four workflows used by various scientific applications and another in which DAGs were synthetically generated. These two cases allow us to study the impact VM provisioning strategies have on real life applications and to further validate the results and extract general rules for arbitrary workflows.

The underlying cloud model was considered to be Amazon EC2 with its eight regions. The prices are listed in Table 1 and were used by Amazon for on-demand instances, with one $BTU = 3,600s$. Communication costs are per GB and were considered only when moving data outside a region. They are applied if the transfer size $\in (1GB, 10TB]$ per month.

The small (**-s*), medium (**-m*), and large (**-l*) instances were considered to have one, two, and four cores, each producing a speed-up of 1, 1.6, and 2.1 times the default one core case. The speed-ups are those reported by the statistical package Stata/MP [4]. One CPU unit is roughly the equivalent of a 1.0-1.2 GHz 2007 Opteron system[5]. For the communication speed we considered small and medium instances to have 1Gb links while the others to have 10Gb links.

For comparison we have also used two known algorithms for workflow scheduling, Gain and CPA-Eager, which similar to *AllPar1LnSdyn*, augment VMs inside a given budget in order to reduce makespan. The initial placement in both cases has been done by using *H-OneVMperTask*. The maximum allowed cost was set to four times respectively twice the cost needed if *H-OneVMperTask-s* would have been used instead.

Overall we considered only strategies that produced both *makespan gain* and *cost savings* with respect to a reference value set to *H-OneVMperTask-s*. Both metrics are defined as percentages of makespan or cost improvement compared to the reference value. Considering only the small instance type, given that this method is optimal in terms of makespan –when boot/shutdown times are considered– we looked for methods that are cheaper than it. We also looked at the possibility of having a faster but cheaper result with a superior VM instance.

Simulations for both real and randomly generated graphs were done by using a custom simulator written in Python. This allowed us to observe the ideal scenario in which no boot/shutdown times exist. In addition in order to check the impact boot/shutdown times in environments such as Amazon EC2 we used a cloud simulator[6] built on top of SimGrid [7]. It allows us to

[4] http://www.stata.com/statamp/statamp.pdf (accessed Jun 24th 2013)

[5] http://aws.amazon.com/ec2/ (accessed Feb 26th 2013)

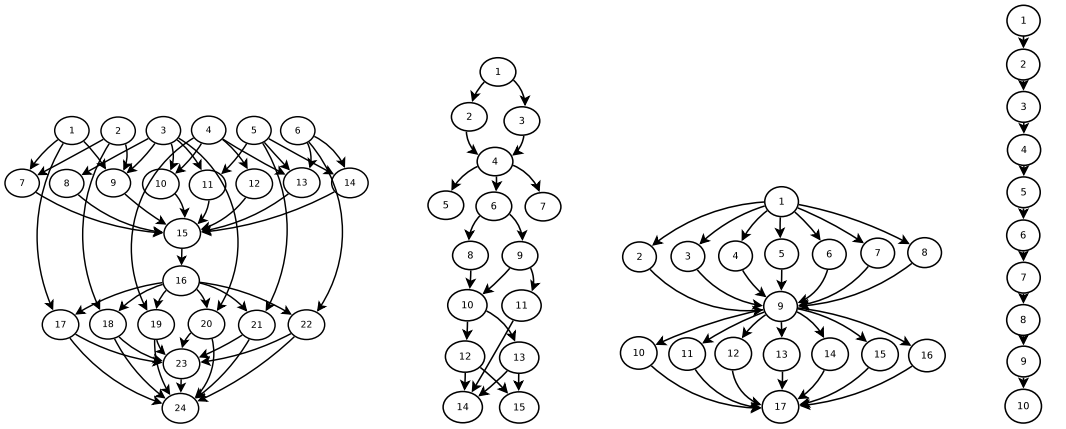
[6] can be downloaded at `https://gforge.inria.fr/projects/simiaas/` together with the four real workflows experimental setup

better model boot/shutdown times and the network traffic required to move the VM. The overhead reported in Sect. 3.2.2 was based on results obtained by using it. Furthermore for the case of real workflows depicted in Sect. 4.1 –because of the small makespan overhead observed when boot/shutdown are considered and since cost remains virtually unchanged– the results without boot/shutdown times stand valid.

4.1 Real Workflows

For this scenario we were particularly interested in the makespan gain/savings ratio when different provisioning strategies are used.

Four different workflows have been used in our tests: Montage [9], CSTEM [10], MapReduce and a simple sequential one. Figure 5 depicts their structure. Figure 6 depicts the results when the Pareto distribution is used.



(a) Montage (b) CSTEM (c) MapReduce (d) Sequential

Figure 5: Structure of the four real workflows.

Montage (cf. Fig. 5(a)) is a workflow used in astronomical image processing. Its size varies depending on the dimension of the studied sky region. In our tests we used a version with 24 tasks. The workflow dependencies are quite intermingled and data dependencies are not only from one level. This type of workflow makes it particularly interesting for locality aware data intensive scheduling. At the same time the large number of parallel tasks makes it good for studying the efficiency of parallel VM provisioning policies.

CSTEM (cf. Fig. 5(b)) is a workflow used in CPU intensive applications. Its relatively sequential nature with a few parallel tasks and several final tasks makes it a good candidate for studying the limits of the parallel VM provisioning policies against the *OneVMperTask* and a particular case of *StartParExceed* in which all tasks of a workflow with a single initial task are scheduled on the same VM.

MapReduce is a model first implemented in Hadoop which aims at processing large amount of data by splitting it into smaller parallel tasks. In Figure 5(c) we depict a version in which there are two sequential map phases. Its original use and the large (variable) amount of parallel tasks it can hold makes it useful in the study of various parallel VM provisioning policies for data intensive tasks and in showing their benefits. The number of tasks on each allocated VM is defined by the used provisioning and allocation policies. For instance in case of *H-OneVMperTask* we always have a single task per VM. As for the rest it depends on the tasks' execution times (**-[Not]Exceed*) and on whether they fit or not the BTU (**-NotExceed*).

The sequential workflow (cf. Fig. 5(d)) is a typical example of a serial application with dependencies, e.g., makefiles. It is the opposite of the parallel intensive MapReduce model. In our paper it is used to show the limits of the parallel provisioning policies and the benefits of using *OneVMperTask* and *StartPar[Not]Exceed*.

Three scenarios have been considered for the execution times.

First a Pareto distribution [11] with a shape of 2 and a scale set to 500. For data transfers a shape of 1.3 has been used. Second one assumes a best case scenario on which all tasks are equal in length and could fit into a single VM. In this case, given n tasks and e the individual

runtime such that $ne \leq BTU$ we have for a sequential provisioning a total number of VMs equal to $m = 1$ (cost=1 BTU) while for a parallel one we have $m = n$ (cost=n BTU).

Finally, a scenario where tasks have equal length and execution times greater than one BTU. We make it exceed one BTU even for the most powerful VM instance type, i.e., $BTU < e_i/2.7 < e$.

Given the reference results provided by *H-OneVMperTask-s* we marked in Fig. 6) in the upper-left corner its values while all the values of interest for us are in the bottom-right square defining the area where both savings and gain occur. The diagonal can be used to asset the faster/cheaper ratio. Anything above it is slower and more expensive than the reference value, while the algorithms below are faster and cheaper. We also searched for any strategy which might provide stable results in terms of cost and makespan throughout the tests.

Based on whether cost or makespan is the target the following can be noticed:

Savings: much more of the tested SAs fall in this category than in the case of makespan. This seems to indicate that given our algorithms cost improvements are more feasible than makespan improvements. The most interesting case is that of *AllPar[Not]Exceed* which produces a stable gain throughout the three cases but the savings fluctuate drastically as noticed in Table 3 (in parenthesis there is the loss inflicted in the case of the Pareto distribution). Using small instances is the only case in which savings are positive (negative loss) while the rest can induce losses depending on the workflow type of up to 40% for the medium respectively 166% for the large instance. While for the medium instance the stable gain of 37% can be a suitable trade-off, the small gain of only 52% noticed for the large instance does not cover the loss which is about three times larger. Thus this provisioning strategy is best suited for small instances while for medium ones only when a trade-off in savings is preferred in order to achieve a better gain.

Furthermore, except the worst case with the sequential workflow, the costs inflicted by the previous two SAs can be further reduced with the *AllPar1LnS* and *AllPar1LnSDyn* algorithms.

Gain: the dynamic SAs seem to be at disadvantage in this case. In addition it seems that this situation is hard to be achieved and strongly depends on the execution times. No SA falls in this situation for the worst case while the best case has the most of them. This can indicate that if gain is the target small execution times are needed for best results. The medium and large instances dominate with the *AllParExceed-m* being a winner for Montage (best case), CSTEM (Pareto case) and MapReduce (best case). It also seems to indicate that for gain to overcome intensively parallel workflows are needed with this SA. Regarding the sequential workflow any SA that considers large instances can be used. In its case it seems that the smaller the execution times the better the results are –i.e., for the best case the gain/savings are balanced.

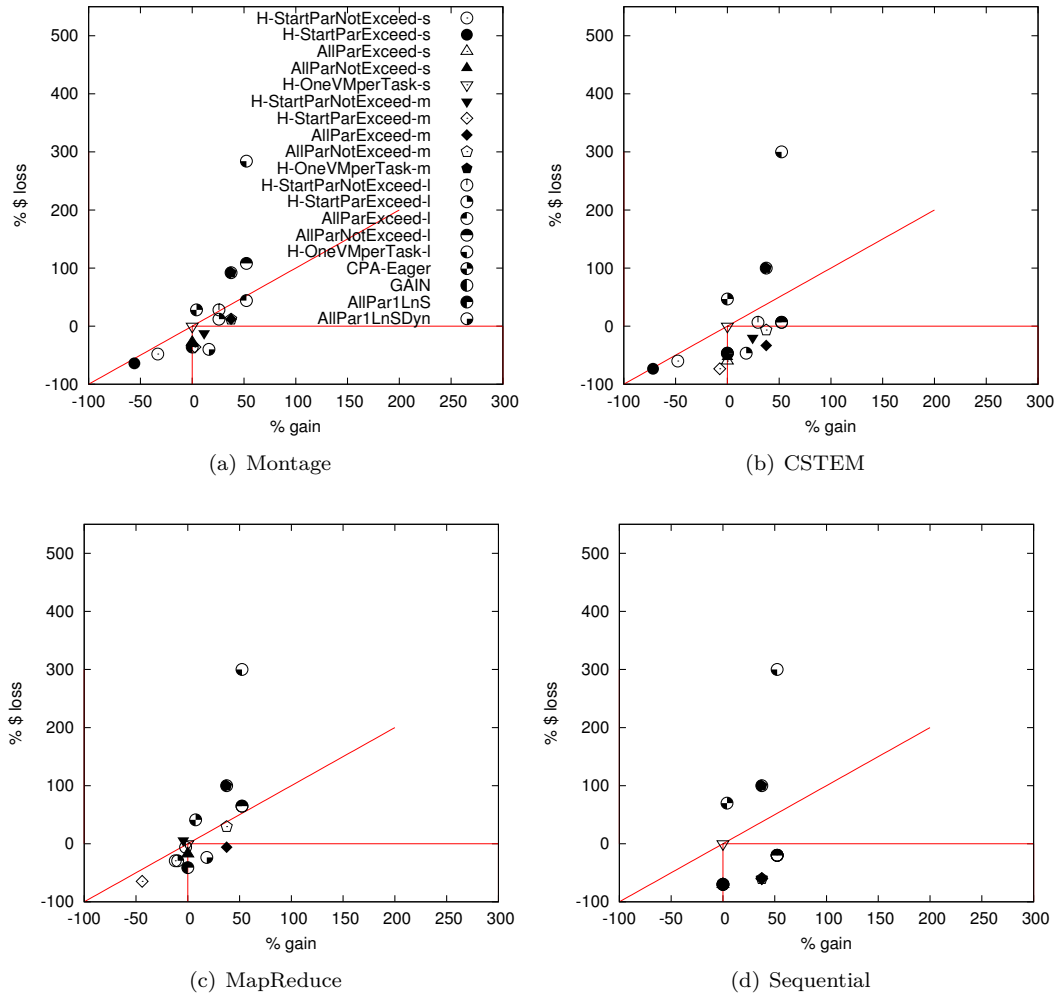
Balanced: When considering a balanced gain/savings the *H-StartPar[Not]Exceed* SAs seem to have an advantage especially in the case of Montage and CSTEM. Depending on the nature of the execution times it can be used with small (worst case) or medium (Pareto case) instances. Their failure to provide similar results for the best case seems to indicate that these algorithms provide good results for large (and heterogeneous) execution times.

Table 3: Savings fluctuation vs. stable gain for AllPar[Not]Exceed.

VM	% loss interval				% loss	% gain
	Montage	CSTEM	MapReduce	Sequential		
s	$[-62, 0]$ (-28)	$[-73, 0]$ (-53)	$[-58, 0]$ (-17)	$[-69, 0]$ (-70)	$[-90, 0]$	0%
m	$[-25, 45]$ (12)	$[-46, 40]$ (-6)	$[-17, 37]$ (29)	$[-80, 33]$ (-60)	$[-80, 40]$	37%
l	$[50, 163]$ (6)	$[6, 155]$ (6)	$[-64, 134]$ (64)	$[-60, 166]$ (-20)	$[-64, 166]$	52%

Looking at the dynamic SAs it can be stated that for *AllPar1LnSDyn* it seems the algorithm's performance is proportional to the heterogeneity of the execution times. This SA is without doubt the only one that manages to remain in the target square for all cases. Nonetheless it appears that it generally produces better savings than gain thus in a gain oriented scenario other solutions should be looked for (cf. Table 4).

Considering our two dynamic algorithms from the literature, Gain and CPA-Eager, we noticed that although they were the most stable, their profit loss ranges between [45, 100]%. The stability comes from the limit on the price and from the fact that they only augment the VMs for essential tasks, e.g., those on the critical path. The cost loss is explicable from the use



(a) Montage (b) CSTEM

(c) MapReduce (d) Sequential

Figure 6: Makespan gain vs. cost loss for CPU intensive workflows

Table 4: Test results conclusion summary for real workflows.

	Savings	Gain	Balance
MapReduce	*AllPar1LnSDyn*	*AllParExceed-m* ⊕ small & heterogeneous tasks	*AllParLnSDyn* ⊕ heterogeneous tasks
Montage	*AllPar1LnSDyn*	*H-StartPar[Not]Exceed-l*, *AllPar[Not]Exceed-m* ⊕ short tasks	*H-StartParNotExceed-[m|s]*, ⊕ heterogeneous respectively long tasks
CSTEM	*AllPar1LnSDyn*	*AllParNotExceed-m* for heterogeneous tasks	*[H-Start|All]ParNotExceed-[s|m]* ⊕ long respectively heterogeneous tasks
Sequential	**-s* and *AllPar1LnSDyn* ⊕ small & heterogeneous tasks	**-l* with heterogeneous tasks	**-l* with short tasks

of the reference method as initial schedule. This will produce in all cases a price falling outside our range of interest. A solution would be to use another algorithm, such as *AllParExceed*, as initial schedule.

While not many SAs can match the gain offered by *H-OneVMperTask-l* its large loss of 200-300% makes it inefficient in face of other SAs that reduce loss or even make profit.

It can be noticed that most of the algorithms present in our target square are made up of either dynamic or small/medium instances. The reason behind this is that the benefit of renting large VMs against the speed gain is of only 0.675 compared to the medium case (=0.8) and the default small one (=1). The provided speed-up is simply too small compared to the price needed to rent such VMs.

Table 4 summarizes the main conclusions drawn from our three experiments. All in all the dynamic *AllPar1LnSDyn* SA can be used in profit oriented scenarios, the medium instance in combination with some SAs can lead to better gain while the large instances prove efficient for sequential workflows when gain is targeted.

4.2 Synthetic Workflows

For the case of randomly generated workflows we used the following approach:

The number of levels was varied between $nl = \overline{1, 12}$ and each level was assigned a variable average number of tasks $tpl = \{2.5, 7.5, 12.5\}$. Each DAG was generated using the *samepred* method [28] with the average number of predecessors being set to 3. For the execution times two scenarios have been considered: normally distributed ($N(150, 50)$, $N(3600, 200)$, $N(4500, 200)$, $N(5400, 200)$, $N(7200, 200)$, $N(9000, 200)$) and Pareto distributed ($P(2, 500)$). The reason for these choices is that we wanted to investigate the impact the execution time distribution has on the scheduling methods. Furthermore, in general, largely parallel applications such as Montage or MapReduce usually exhibit similar execution times for task on the same level of parallelism, which is better modeled through a normal distribution than a Pareto one. However non-deterministic (in terms of execution times) workflows such as those solving complex mathematical problems –where the execution time of a problem depends on the method chosen at runtime by the software– could present large tasks run in parallel with smaller ones.

Given the above setup 5, different DAGs are generated for each $\{nl, tpl\}$ configuration. For each DAG, 5 different experiments are run in order to vary the execution times of each task, leading to a total number of 175 experiments;

Two cases have been assumed: DAGs with few (long workflows) respectively many tasks per level (largely parallel). For the former we assumed an average number of 2.5 tasks while for the latter all the remaining values in the *tpl* set were used;

For each experiment we counted the percentage of experiments when an algorithm is among the best. To relax the condition we selected algorithms that were close to the best within 10% of makespan and cost. To be taken into consideration algorithms had to be better (but not best) in both cost and makespan than the reference *OneVMperTask-s* algorithm. This is to ensure that algorithms that are the best for only one objective still reduce cost as compared to the reference one.

Given the numerous scenarios we considered in this case and for reasons related to space, we present in what follows only the main conclusions instead of detailing each result individually.

Our initial expectations were to discover a single dominant VM provisioning method for each individual use case. Results however showed that when small execution times –either normal or Pareto shaped– there is no clear dominant algorithm that achieves a success rate above 90%, especially when worklows with many parallel levels and tasks are considered. The reason could be in the combination between the fine-grained heterogeneous execution times and the various workflow structures that were used. In contrast as execution times get larger –than one BTU– the dominant algorithms have success rates placed regularly above 90%.

In addition it has been observed that as far as cost is concerned the number of dominant algorithms tends to be smaller than for the case of the makespan. For $nl = 1$ all dominant algorithms achieve high success rates (around 90% or more) by relying on mostly small instances. When $nl > 1$ and execution times are above one BTU there is a single dominant algorithm. Since in most cases the algorithm is *AllParExceed-s* the reason could lie in the combination of parallelism and sequential execution on already rented VMs and in the fact that faster instance types do not bring the same cost reduction as compared to makespan gain which could place them outside the area of interest –as previously mentioned in Sect. 4. Except *AllPar[Not]Exceed-s* and *OneVMperTask-s* all other algorithms tend to reduce the number of VMs and thus possibly increase the makespan above the reference value. Concerning medium and large instance based algorithms it seems that the inflicted cost is greater than the speed-up gained, probably even exceeding the reference value. So it is not unlikely to end up with the *AllParExceed-s* as the single best algorithm in terms of cost.

Regarding the two objectives of makespan and cost we can make the following comments summarized in Tables 5 and 6. Figures 7, 8, 9 and 10 present the percentage of cases a particular algorithm has provided the best solution in terms of makespan or cost.

When considering **makespan** alone, we observe that for $nl = 1$ and workflows with few tasks (e.g., $tpl = 2.5$) the *-s* algorithms (except *OneVMforAll-s*) seem to provide the best value except an isolated case of execution times situated between one and two BTUs where the *-m* algorithms (except *OneVMforAll-m*) are also dominant. A similar behavior was noticed for largely parallel workflows except in the case of the Pareto distribution (cf. Fig. 7) when *AllPar1LnSDyn* was dominant although at a steadily decreasing efficiency (100% for $tpl = 2.5$ downto 59% for $tpl = 12$). Its efficiency in these cases could be explained by the Pareto shaped distribution which allows faster longer tasks to be executed in parallel with many smaller parallel tasks running on the same VM.

For $nl > 1$ a rise in the efficiency of some algorithms using medium or large instances is observed, especially when $e_i < 1BTU$ (cf. Fig. 7), probably due to their capacity of reducing the makespan enough for the cost to drop below the reference value. For $nl = 1$ we noticed that using small instances provides the best results, except an interval between 1 BTU and 1.6 BTUs or exactly the speed-up of the medium instance (cf. Fig. 8). The motivation could be that this particular interval allows for the speed-up inflicted by the medium instances to reduce costs sufficiently as to surpass the reference value. Since the gain in makespan is far greater than that of the small instances these methods therefore replace the **-s* ones as dominant. While we should expect a similar result for larger values –comparable with the speed-up induced by the large instance– results proved otherwise. The reason could be in the cost loss/makespan gain ratio which is around 3 (cf. Table 3) making costs simply to high to fit our desired interval below the reference threshold.

Results showed that as execution times increase –above 1 BTU– the success rate of the dominant algorithms stabilizes and achieves values close to 100%. This means that for these cases we can safely pick one algorithm and use it without being restricted by the probability factor as in the case of small execution times; i.e., the Pareto and Normal(150,50) distributions.

Table 5: Makespan results for randomly generated DAGs.

e_i	$nl = 1$		$nl > 1$	
	less parallelism	much parallelism	less parallelism	much parallelism
Pareto	**-s* except *OneVMforAll-s*	*AllPar1LnSDyn*	*AllParExceed-l*	*AllPar1LnS-m*, *AllPar1LnS-l*
short	**-s* except *OneVMforAll-s*	**-s* except *OneVMforAll-s*	*AllParExceed-l*, *AllPar1LnS-l*	*AllPar1LnS-m*
large	*-s* except *OneVMforAll-s*; *-m* except *OneVMforAll-m* ($e_i \in (3600, 7200)$)	*AllPar1LnSdyn*, *-s* except *OneVMforAll-s*; *-m* except *OneVMforAll-m* ($e_i \in (3600, 7200)$)	*AllParExceed-m*; *-m* except *OneVMforAll-m* ($e_i \in (3600, 7200)$)	*AllPar1LnSdyn*, *-s* except *OneVMforAll-s*; – *-m* except *OneVMforAll-m* ($e_i \in (3600, 7200)$)

Concerning **cost** we noticed that dominant algorithm is *AllParExceed-s* except for the Pareto and N(150,50) distributions when *AllPar1LnS-s* and *AllPar1LnSdyn* tend to dominate (cf. Fig. 9). The reason could be due to the shape of the distribution which allows for parallelism reduction and instance augmentation. *AllPArExceed-s* on the other hand allows for parallelism exploitation and also cost minimization by permitting tasks to exceed their BTU and thus avoid renting new ones and wasting both idle time and costs. Interestingly enough the same scenario when the best results for makespan when the execution times is between 1 and 1.6 BTUs repeated here. The difference is that for costs it only applies when $nl = 1$ (cf. Fig. 10). The reason lies in the fact that since we have parallel tasks that are larger than one BTU all medium instances, due to their speed-up will reduce the execution time to at most one BTU which is the smallest payable unit. Thus we obtain a cost identical to smaller to that of reference value and at the same time faster.

Concluding, using larger instance types such as medium sized can prove an advantage due to either heterogeneity in execution times or cost/makespan gain when small execution times

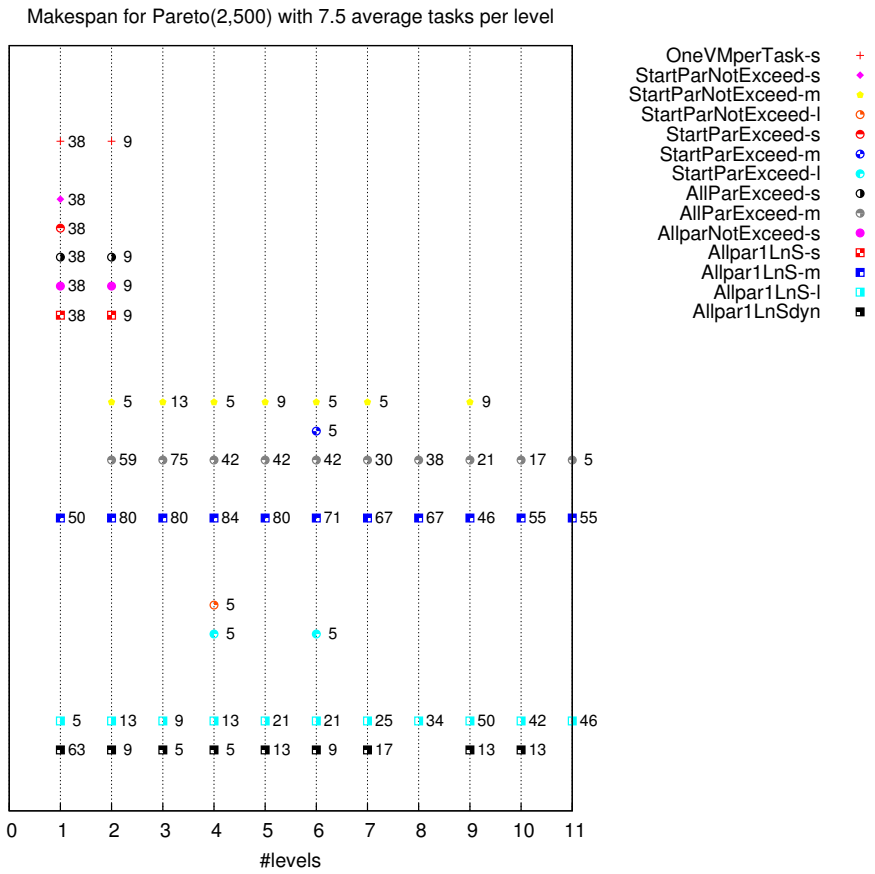


Figure 7: Best algorithm % for makespan for Pareto(2,500) and $tpl = 7.5$.

are used (below one BTU) or the execution time is between one and 1.6 BTUs. In this case the medium instances also provide a balanced cost/makespan minimization. Dynamic algorithms such as *AllPar1LnS-s* or *AllPar1LnSdyn* have a limited efficiency especially for large tasks being overcome both in terms of makespan and cost by others. In addition, although for small execution times a probabilistic choice based on past data between algorithms is required, for large execution times the dominant algorithms for either makespan or cost can be selected with a probability of one due to their high success rates. The same is applicable also for cost alone when $nl = 1$ and execution times are large. Finally for large execution times where *-s* algorithms produce the best makespan, if a balanced cost/makespan minimization is desired, the *AllParExceed-s* can be used.

5 Conclusions

Most work related to workflow scheduling algorithms for clouds has focused on extending existing grid oriented algorithms to clouds by renting whenever necessary extra cloud resources. These consider clouds as extensions to the local resources. While others have focused on building fully cloud oriented algorithms, none has investigated the impact choosing the correct provisioning policy has on the schedule. This is especially important when adapting existing grid algorithms to the cloud environment by adding a suitable provisioning policy.

As shown in a previous paper [12], workflow structure, tasks size, and used VM instance type, all influence the results. It is thus necessary to derive a mechanism for dynamically adapting the SAs based on this information. This paper extends those results and provide a mathematical model and some general guidelines for randomly shaped workflows. To do this, we first needed to formally model makespan and cost –the two objectives we consider in this paper– for the cases where single and multiple VM instance types are used. As VM boot/shutdown times also play an important role as they are paid for but cannot be actually used, we included them in our model as well. The model allows a first selection based on mathematical formulas. Whenever this approach is not sufficient, then the results of our tests can be used to determine, based on workflow and task characteristics, which VM instance type and provisioning method

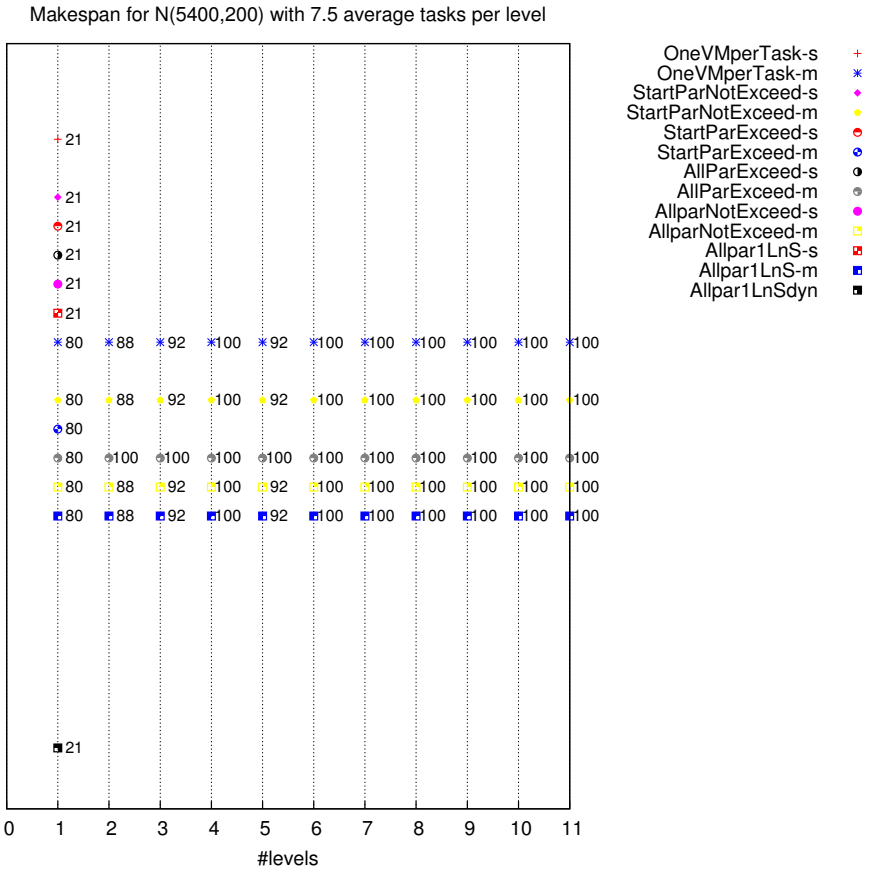


Figure 8: Best algorithm % for makespan for Normal(5400,200) and $tpl = 7.5$.

Table 6: Cost results for randomly generated DAGs.

e_i	$nl = 1$		$nl > 1$	
	less parallelism	much parallelism	less parallelism	much parallelism
Pareto	*AllPar1LnS-s*, *AllPar1LnSdyn*	*AllPar1LnS-s*	*AllParExceed-s*	*AllPar1LnS-s*
short	*AllPar1LnS-s*, *AllPar1LnSdyn*	*AllPar1LnS-s*	*AllPar1LnS-s*, *AllPar1LnSdyn*	*AllPar1LnS-s*
large	*StartParExceed-s*, *AllParExceed-s*, *AllParNotExceed-s*, *AllPar1LnS-s*, *AllPar1LnSdyn* and *OneVMperTask-m* $\oplus$ their *-m* equivalents for $e_i \in (3600, 7200)$	*H-StartParExceed-s*, *AllParExceed-s*, *AllParNotExceed-s*, *AllPar1LnS-s*, *AllPar1LnSdyn* and *OneVMperTask-m* $\oplus$ their *-m* equivalents for $e_i \in (3600, 7200)$	*AllParExceed-s*	*AllParExceed-s*

to use in order to minimize makespan or costs.

In order to take full advantage of the multi-core property of most VM instance types offered

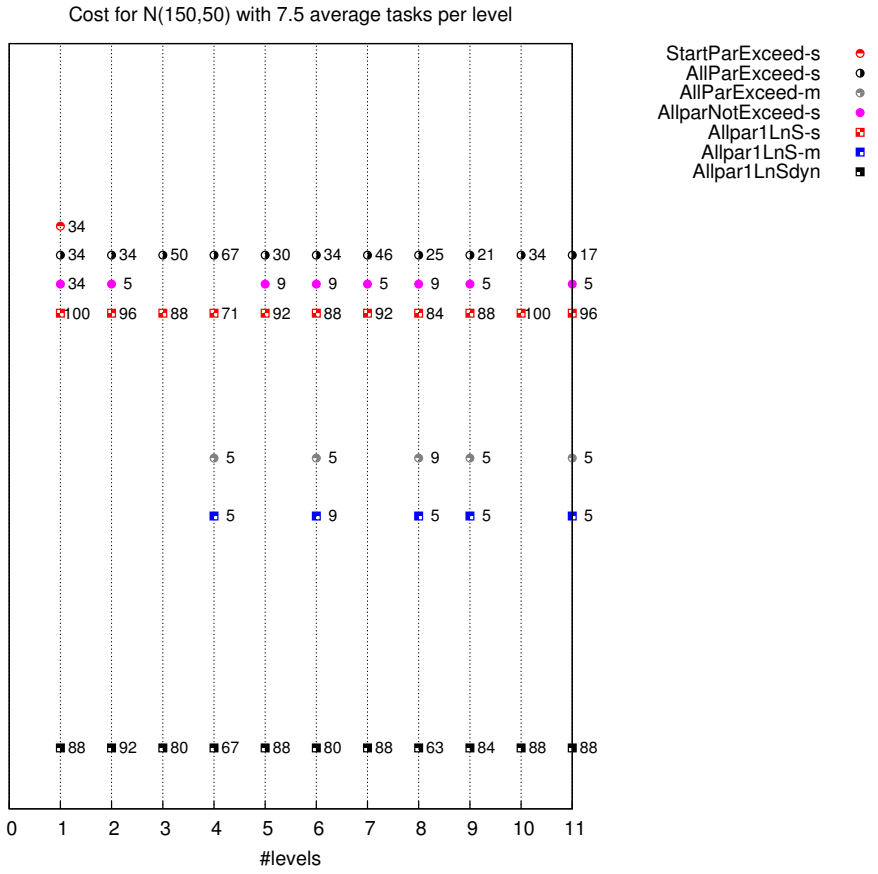


Figure 9: Best algorithm % for cost for Normal(150,50) and $tpl = 7.5$.

by public cloud providers we considered that each workflow task is multi-threaded capable. Some remarks for how our model can be adapted for single-threaded tasks have also been given.

Test results have shown that, given a speed-up which is smaller than the price increase when choosing a faster instance type, faster instance types are suited only for certain execution times, and usually for tasks with many levels of parallelism. As cost is higher than gain most medium and large instances fall outside of our targeted area of improving both objectives as compared to the reference one. An exception seems to occur when sequential workflows with heterogeneous times are used. In this case large instances provide the best gain at a price lower than the reference one with small instances. Dynamic algorithms that target parallelism reduction and instance type augmentation –i.e., *AllPar1LnS*, *AllPar1LnSDyn*– are also inefficient for large execution times that exhibit little heterogeneity (e.g., the map tasks in a MapReduce application). For small ($< 1BTU$) and heterogeneous tasks we noticed that the dominant algorithms is not clearly differentiated as in the case of large and more homogeneous tasks. This means that, in this case, choosing between provisioning methods would require a probabilistic approach if any of the two objectives is targeted. Contrary, for the latter case, the *AllParExceed-s* algorithm proved to be most successful in minimizing both cost and makespan.

Overall reducing both cost and makespan at the same rate is difficult with most algorithms being optimized for at most one objective. For some scenarios however *AllParExceed-s* and several *-m* versions are capable of minimizing both equally.

Finally our tests using constant boot/shutdown times, that are independent on the number of parallel booted VMs, indicated no significant difference with regard to the ideal case where these times are set to zero. While this could be different for other cases, we emphasize the relevance of these results as they correspond to the setup of one of the most popular cloud provider, i.e., Amazon.

Further work aims at integrating results obtained from this research in a comprehensive knowledge base that can be later used by an adaptive scheduler relying on (un)supervised learning techniques. This will allow automatic optimization decisions independent on workflow characteristics. An alternative approach would be to asses the outcomes of each algorithm given the actual workflow to execute right before its submission in the system. The efficiency

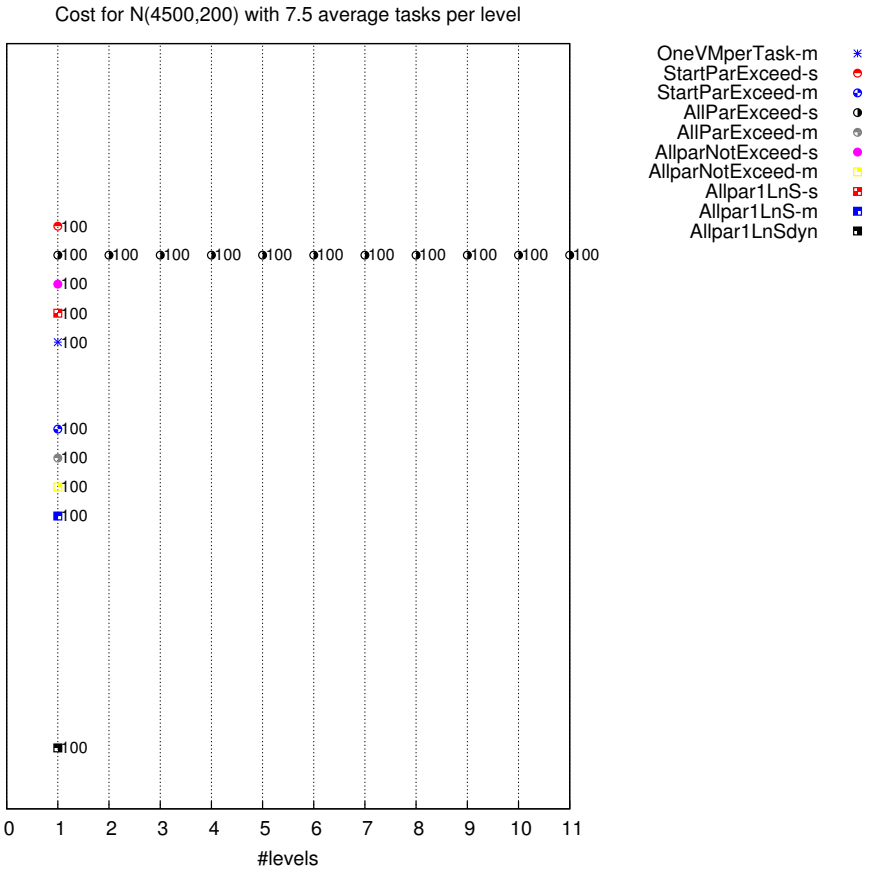


Figure 10: Best algorithm % for cost for Normal(4500,200) and $tpl = 7.5$.

of such a method is also under consideration. Finally we plan on investigating to what extent our model can be adapted to arbitrarily shaped workflows.

Acknowledgements This work has been partially supported by the French ANR project SONGS 11-INFRA-13.

References

[1] Bittencourt, L., Madeira, E.: Hcoc: a cost optimization algorithm for workflow scheduling in hybrid clouds. Journal of Internet Services and Applications **2**, 207–227 (2011). URL `http://dx.doi.org/10.1007/s13174-011-0032-0`. 10.1007/s13174-011-0032-0

[2] Bittencourt, L.F., Madeira, E.R.M.: A performance-oriented adaptive scheduler for dependent tasks on grids. Concurr. Comput. : Pract. Exper. **20**(9), 1029–1049 (2008). DOI 10.1002/cpe.v20:9. URL `http://dx.doi.org/10.1002/cpe.v20:9`

[3] Bobroff, N., Kochut, A., Beaty, K.: Dynamic placement of virtual machines for managing sla violations. In: 10th IFIP/IEEE International Symposium on Integrated Network Management, pp. 119–128. IEEE (2007). URL `http://dblp.uni-trier.de/db/conf/im/im2007.html#BobroffKB07`

[4] den Bossche, R.V., Vanmechelen, K., Broeckhove, J.: Cost-optimal scheduling in hybrid IaaS clouds for deadline constrained workloads. In: IEEE CLOUD, pp. 228–235 (2010)

[5] Byun, E.K., Kee, Y.S., Kim, J.S., Maeng, S.: Cost optimized provisioning of elastic resources for application workflows. Future Generation Computer Systems **27**(8), 1011 – 1026 (2011). DOI 10.1016/j.future.2011.05.001. URL `http://www.sciencedirect.com/science/article/pii/S0167739X11000744`

[6] Caron, E., Desprez, F., Muresan, A., Suter, F.: Budget constrained resource allocation for non-deterministic workflows on an iaas cloud. In: Y. Xiang, I. Stojmenovic, B. Apduhan, G. Wang, K. Nakano, A. Zomaya (eds.) Algorithms and Architec-

tures for Parallel Processing, *Lecture Notes in Computer Science*, vol. 7439, pp. 186–201. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-33078-0_14. URL http://dx.doi.org/10.1007/978-3-642-33078-0_14

[7] Casanova, H., Legrand, A., Quinson, M.: Simgrid: a generic framework for large-scale distributed experiments. In: Proceedings of the Tenth International Conference on Computer Modeling and Simulation, UKSIM '08, pp. 126–131. IEEE Computer Society, Washington, DC, USA (2008). DOI 10.1109/UKSIM.2008.28. URL http://dx.doi.org/10.1109/UKSIM.2008.28

[8] Deelman, E., Singh, G., Livny, M., Berriman, G.B., Good, J.: The cost of doing science on the cloud: the montage example. In: SuperComputing'08, p. 50 (2008)

[9] Deelman, E., Singh, G., Su, M.H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G.B., Good, J., Laity, A., Jacob, J.C., Katz, D.S.: Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Sci. Program. **13**(3), 219–237 (2005). URL http://dl.acm.org/citation.cfm?id=1239649.1239653

[10] Doğan, A., Özgüner, F.: Biobjective scheduling algorithms for execution time-reliability trade-off in heterogeneous computing systems*. Comput. J. **48**(3), 300–314 (2005). DOI 10.1093/comjnl/bxh086. URL http://dx.doi.org/10.1093/comjnl/bxh086

[11] Feitelson, D.G.: Workload modeling for computer systems performance (2013). URL http://www.cs.huji.ac.il/~feit/wlmod/. Version 0.43

[12] Frincu, M., Genaud, S., Gossa, J.: Comparing provisioning and scheduling strategies for workflows on clouds. In: Workshop Procs. of 28th IEEE Int. Parallel & Distributed Processing Symposium, pp. 2101–2110. IEEE (2013)

[13] Frîncu, M.E.: Scheduling highly available applications on cloud environments. Future Generation Computer Systems **32**(0), 138–153 (2014). DOI 10.1016/j.future.2012.05.017

[14] Google: Google compute engine pricing. URL https://cloud.google.com/pricing/compute-engine. Https://cloud.google.com/pricing/compute-engine (accessed June 20th 2013)

[15] Gu, J., Hu, J., Zhao, T., Sun, G.: A new resource scheduling strategy based on genetic algorithm in cloud computing environment. JCP pp. 42–52 (2012)

[16] Gutierrez-Garcia, J.O., Sim, K.M.: A family of heuristics for agent-based elastic cloud bag-of-tasks concurrent scheduling. Future Generation Computer Systems **29**(7), 1682–1699 (2012). DOI 10.1016/j.future.2012.01.005

[17] Hwang, E., Kim, K.H.: Minimizing cost of virtual machines for deadline-constrained mapreduce applications in the cloud. In: Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on, pp. 130 –138 (2012). DOI 10.1109/Grid.2012.19

[18] Lin, C., Lu, S.: Scheduling scientific workflows elastically for cloud computing. In: Cloud Computing (CLOUD), 2011 IEEE International Conference on, pp. 746 –747 (2011). DOI 10.1109/CLOUD.2011.110

[19] Liu, K.: Scheduling algorithms for instance-intensive cloud workflows. Ph.D. thesis, University of Swinburne Australia (2009)

[20] Lucas-Simarro, J.L., Moreno-Vozmediano, R., Montero, R.S., Llorente, I.M.: Scheduling strategies for optimal service deployment across multiple clouds. Future Generation Computer Systems (2012). DOI 10.1016/j.future.2012.01.007. Accepted proof

[21] Mao, M., Humphrey, M.: Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pp. 49:1–49:12. ACM, New York, NY, USA (2011). DOI 10.1145/2063384.2063449. URL http://doi.acm.org/10.1145/2063384.2063449

[22] Mao, M., Humphrey, M.: A performance study on the vm startup time in the cloud. In: IEEE CLOUD'12, pp. 423–430 (2012)

[23] Michon, E., Gossa, J., Genaud, S.: Free elasticity and free CPU power for scientific workloads on IaaS Clouds. In: 18th IEEE International Conference on Parallel and Distributed Systems. IEEE, Singapour, Singapore (2012). URL http://hal.inria.fr/hal-00733155

[24] Mohammadi Fard, H., Prodan, R., Fahringer, T.: A truthful dynamic workflow scheduling mechanism for commercial multi-cloud environments. Parallel and Distributed Systems, IEEE Transactions on **PP**(99), 1 (2012). DOI 10.1109/TPDS.2012.257

[25] Pandey, S., Wu, L., Guru, S., Buyya, R.: A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In: Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on, pp. 400 –407 (2010). DOI 10.1109/AINA.2010.31

[26] Radulescu, A., van Gemund, A.: A low-cost approach towards mixed task and data parallel scheduling. In: Parallel Processing, International Conference on, 2001., pp. 69 –76 (2001). DOI 10.1109/ICPP.2001.952048

[27] Sakellariou, R., Zhao, H., Tsiakkouri, E., Dikaiakos, M.D.: Scheduling workflows with budget constraints. In: in Integrated Research in Grid Computing, S. Gorlatch and M. Danelutto, Eds.: CoreGrid series. Springer-Verlag (2007)

[28] Tobita, T., Kasahara, H.: A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. Journal of Scheduling **5**(5), 379–394 (2002). DOI 10.1002/jos.116. URL http://dx.doi.org/10.1002/jos.116

[29] Tordsson, J., Montero, R.S., Moreno-Vozmediano, R., Llorente, I.M.: Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers. Future Gener. Comput. Syst. **28**(2), 358–367 (2012). DOI 10.1016/j.future.2011.07.003. URL http://dx.doi.org/10.1016/j.future.2011.07.003

[30] Villegas, D., Antoniou, A., Sadjadi, S.M., Iosup, A.: An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds. In: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), CCGRID '12, pp. 612–619. IEEE Computer Society, Washington, DC, USA (2012). DOI 10.1109/CCGrid.2012.46. URL http://dx.doi.org/10.1109/CCGrid.2012.46

[31] Wu, Z., Liu, X., Ni, Z., Yuan, D., Yang, Y.: A market-oriented hierarchical scheduling strategy in cloud workflow systems. The Journal of Supercomputing **63**, 256–293 (2013). URL http://dx.doi.org/10.1007/s11227-011-0578-4

[32] Zaman, S., Grosu, D.: Combinatorial auction-based allocation of virtual machine instances in clouds. Journal of Parallel and Distributed Computing **73**(4), 495 – 508 (2013). DOI 10.1016/j.jpdc.2012.12.006. URL http://www.sciencedirect.com/science/article/pii/S0743731512002870

[33] Zhao, H., Sakellariou, R.: An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. In: H. Kosch, L. BÃ¶szÃ¶rmÃ©nyi, H. Hellwagner (eds.) Euro-Par 2003 Parallel Processing, *Lecture Notes in Computer Science*, vol. 2790, pp. 189–194. Springer Berlin Heidelberg (2003). DOI 10.1007/978-3-540-45209-6_28. URL http://dx.doi.org/10.1007/978-3-540-45209-6_28

Appendix: Theoretical Proofs

Proposition 1:

Proof. We assume three tasks and test the cases where they are either parallel or sequential tasks independently:

Parallel tasks: let there $cost_2$ be the cost of another method such that $cost_2 < cost_s$. From Relation 3 we have that $cost_2 < \lceil \frac{e_1+e_2+e_3}{BTU} \rceil$. For $cost_2$ we have that either all three tasks are scheduled on different VMs or two are put on a single one and the third on a second one. For the former case we get $cost_2 = \lceil \frac{e_1+e_2}{BTU} \rceil + \lceil \frac{e_3}{BTU} \rceil$. For the latter we have $cost_2 = \lceil \frac{e_1}{BTU} \rceil + \lceil \frac{e_2}{BTU} \rceil + \lceil \frac{e_3}{BTU} \rceil$. In both case given that $\lceil \sum \cdot \rceil \leq \sum \lceil \cdot \rceil$ we obtain a contradiction.

Sequential tasks: the proof is similar to that for parallel tasks. □

Proposition 2:

Proof. Straightforward. As it allocates one VM for each task we end up with as many VMs as tasks irrespective on the VM instance type. □

Remark 1:

Proof. Parallel tasks: the makespan for *OneVMperTask* is $\max e_i$ (cf. Relation 2). We assume three parallel tasks. They can be placed in the following configurations: each on its VM, two tasks on one VM and the other on another, or all three tasks on a single VM. Given their parallelism case we obtain that the makespan is equal to: $\max e_i$ for the first; $\max(e_i + e_j, e_k)$ (where $\{i, j, k\}$ represents a permutation of $\{1, 2, 3\}$) for the second; and $\sum e_i$ for the third. It is easily proven that $\max e_i$ is less or equal with any of the three cases.

Sequential tasks: in this case a task has to finish before its successor can start the makespan for a task n can be recursively written as $\sum_{i=1}^{n-1} + e_n$ which leads to the same formula as for *OneVMperTask*: $\sum e_i$. □

Remark 2:

Proof. Similar with the proof for sequential tasks for Remark 1. □

Proposition 3:

Proof. The equalities are straighforward given the definitions of each method. All method will schedule each task on an independent VM. Totprove that $cost_p \geq cost_s$ we use Relations 1 and 3 and the property of the ceil function: $\lceil \sum \cdot \rceil \leq \sum \lceil \cdot \rceil$. □

Proposition 4:

Proof. Sketch: the proof is similar to the one for Proposition 3. Since there is no task parallelism methods like *AllParExceed*, *StartParExceed* schedule all the tasks on a single VM. This is the same as for *OneTaskforAll*. On the other side *OneVMperTask*, *AllParNotExceed* and *StartParNotExceed* rent more VMs due to their policies. Hence the first rents one VM for each task giving a cost cf. Rel. 1. While it is possible for the other two to provide the same cost if the sum of any two consecutive tasks is greater than a BTU in general it is possible to schedule at least two tasks on the same VM. This reduces the cost for these by one BTU as compared to *OneVMperTask*, but still keeps it above the one offered by *OneVMforAll* which is the optimal. The only exception takes place when $\sum e_i \leq BTU$ in which case all methods except *OneVMperTask* give the same costs. □

Proposition 5:

Proof. Sketch: The first two equalities are trivial. As all methods take full advantage of parallelism there is no deterioration in makespan and thus all give the same results as *OneVMperTask*. The *StartParNotExceed* produces a makespan at least equal to that of *AllParNotExceed* since it schedules sequentially parallel tasks if their number exceeds that of the initial tasks. The only scenario in which they would provide the same result is a scenario in which the sum of any two tasks is always greater than a BTU, hence forcing *StartParNotExceed* to allocate a new VM for each task. *StartParExceed* is even worse as the number of parallel running VMs depends on the number of initial tasks. If the maximum number of parallel tasks in the workflow does not exceed this value than the results are identical to those provided by *StartParNotExceed* and *AllParNotExceed*. In the worst case it will schedule all tasks on a single VM which is identical to the *OneVMforAll* method. □

Proposition 6:

Proof. Sequential tasks are chain workflows in which each task depends only on at most one predecessor. For this reason the makespan is always equal with $\sum e_i$. Since this particular workflow does not exhibit any parallelism in it all provisioning methods built to take it into consideration do not offer any makespan optimization. Given Fig. 3 we easily notice that all of them become equivalent to *OneVMforAll* with the makespan given by Rel. 4. □

Proposition 7:

Proof. We prove each inequality individually.

Considering the first inequality, for $cost_{AllParNotExceed}$ we only need to investigate the case in which a new task i needs to be placed on a VM already running another task. We get two scenarios: one in which task i will not exceed the remaining BTU on a selected VM and can thus be scheduled on it; and another one in which it exceeds the remaining time and requires a new VM. For the first scenario we have that $cost_{AllParNotExceed} = \lceil cost_{current} \rceil + \lceil \frac{e_i}{BTU} \rceil$ which is identical to $cost_p = \lceil cost_{current} \rceil + \lceil \frac{e_i}{BTU} \rceil$. For the second we obtain $cost_{AllParNotExceed} = \lceil cost_{current} + \frac{e_i}{BTU} \rceil$. Since for this case $cost_p = \lceil cost_{current} \rceil + \lceil \frac{e_i}{BTU} \rceil$ and we know that $\lceil \sum \cdot \rceil \leq \sum \lceil \cdot \rceil$ the inequality follows immediately.

To prove $cost_{AllParNotExceed} \geq cost_{AllParExceed}$ we consider a case in which a new task i exceeds the remaining BTU time and so needs to be scheduled on a new VM –when considering *AllParNotExceed*. In this case we obtain $cost_{AllParNotExceed} = \lceil cost_{current} \rceil + \lceil \frac{e_i}{BTU} \rceil \geq cost_{AllParExceed}$.

To prove $cost_{AllParExceed} \geq cost_{StartParExceed}$ we consider a single initial task and two new parallel tasks i and j following it. We get $cost_{AllParExceed} = \lceil cost_{current} + e_i \rceil + \lceil e_j \rceil$ and $cost_{StartParExceed} = \lceil cost_{current} + e_i + e_j \rceil$. The inequality follows immediately.

The proof that $cost_{StartParExceed} \geq cost_s$ is straightforward. Considering two initial tasks $i1$, $i2$ and three new tasks j, k, l depending on them, we obtain $cost_{StartParExceed} = \lceil cost_{i1} + e_j + e_k \rceil + \lceil cost_{i2} + e_l \rceil$ which is greater or equal to $cost_s = \lceil cost_{i1} + cost_{i2} + e_j + e_k + e_l \rceil$.

Finally the proof that $cost_p \geq cost_{StartParNotExceed} \geq cost_{StartParExceed}$ is similar to that for $cost_p \geq cost_{AllParNotExceed} \geq cost_{AllParExceed}$. □

Proposition 8:

Proof. Immediately considering one initial task and two tasks depending on it with $\sum e_i < BTU$ and $\sum e_i > BTU$. □

Proposition 9:

Proof. The case $\sum e_i \leq BTU$ follows directly by applying Relations 6 and 8. We obtain: $\underbrace{\gamma_k}_{cost_s^k} < \underbrace{\gamma_{k+1}}_{cost_s^{k+1}} \leq \underbrace{n\gamma_k}_{cost_p^k} < \underbrace{n\gamma_{k+1}}_{cost_p^{k+1}}$.

For the general case we know by Rels. 5 and 7 that $\boxed{\inf cost_s^k = \inf cost_p^k}$ and $\boxed{\sup cost_s^k < \sup cost_p^k}$. The proof follows a straightforward approach by comparing the ratio of each relevant combination of sup and inf values. This ratio can be written as a product $f \cdot g$ where g changes depending on the ratio.

We first assume $f < 1$:

$\boxed{\frac{\inf cost_s^{k+1}}{\inf cost_s^k} < 1} = f\frac{\sum e_i}{\sum e_i}$. The proof follows immediately since $f < 1$.

The rest of the relations depend on the value of the BTU:

$\boxed{\frac{\sup cost_p^{k+1}}{\sup cost_p^k} \geq 1} = f\frac{\sum e_i + n\alpha_{k+1}BTU}{\sum e_i + n\alpha_k BTU}$. We obtain that $BTU \leq \frac{(f-1)\sum e_i}{n(\alpha_k - f\alpha_{k+1})}$. As $BTU > 0$ the ratio must also be positive meaning that $\alpha_k - f\alpha_{k+1} < 0$ which is always true since $1 < \frac{\gamma_{k+1}}{\gamma_k}$.

$\boxed{\frac{\sup cost_s^{k+1}}{\sup cost_p^k} \geq 1} = f\frac{\sum e_i + \alpha_{k+1}BTU}{\sum e_i + n\alpha_k BTU}$. The proof is similar to the previous case except that we get an additional condition: $n < \frac{\gamma_{k+1}}{\gamma_k}$ for $BTU \leq \frac{(f-1)\sum e_i}{n\alpha_k - f\alpha_{k+1}}$.

$\boxed{\frac{\sup cost_p^{k+1}}{\inf cost_s^k} \leq 1} = f\frac{\sum e_i + n\alpha_{k+1}BTU}{\sum e_i}$. The proof is similar to the previous case but $BTU \leq \frac{(1-f)\sum e_i}{nf\alpha_{k+1}}$.

We now assume $f > 1$:

$\boxed{\frac{\inf cost_s^{k+1}}{\inf cost_s^k} > 1} = f\frac{\sum e_i}{\sum e_i}$.

$\boxed{\frac{\sup cost_s^{k+1}}{\sup cost_s^k} > 1} = f\frac{\sum e_i + \alpha_{k+1}BTU}{\sum e_i + \alpha_k BTU}$. Since $\alpha_{k+1} \geq \alpha_k$ we obtain our inequality.

$\boxed{\frac{\sup cost_p^{k+1}}{\sup cost_p^k} > 1} = f\frac{\sum e_i + n\alpha_{k+1}BTU}{\sum e_i + n\alpha_k BTU}$. Since $\alpha_{k+1} \geq \alpha_k$ we obtain our inequality.

$\boxed{\frac{\sup cost_s^{k+1}}{\sup cost_p^k} \geq 1} = f\frac{\sum e_i + \alpha_{k+1}BTU}{\sum e_i + n\alpha_k BTU}$. The inequality holds only if $BTU \leq \frac{(f-1)\sum e_i}{n\alpha_k - f\alpha_{k+1}}$.

$\boxed{\frac{\inf cost_p^{k+1}}{\sup cost_p^k} \geq 1} = f\frac{\sum e_i}{\sum e_i + n\alpha_{k+1}BTU}$. The proof is similar to the previous case but $BTU \leq \frac{(f-1)\sum e_i}{n\alpha_k}$.

We now assume $f = 1$:

$\boxed{\frac{\inf cost_p^k}{\inf cost_p^{k+1}} = 1}$ and $\boxed{\frac{\inf cost_s^k}{\inf cost_s^{k+1}} = 1}$. Immediately from Rels. 5 and 7.

□

Proposition 10:

Proof. Follows immediately from Relations. 6 and 8. □

Proposition 11:

Proof. Inequalities $makespan_p^k \leq makespan_s^k$ and $makespan_p^{k-1} \leq makespan_s^{k-1}$ are trivial from Relations 6 and 8.

To prove $makespan_s^k \leq makespan_p^{k-1}$ we must have $\frac{\alpha_{k-1}\sum e_i}{\alpha_k \max e_i} \leq 1$. This happens iff $\frac{\alpha_k}{\alpha_{k-1}} \geq \frac{\sum e_i}{\max e_i}$. □

Proposition 13:

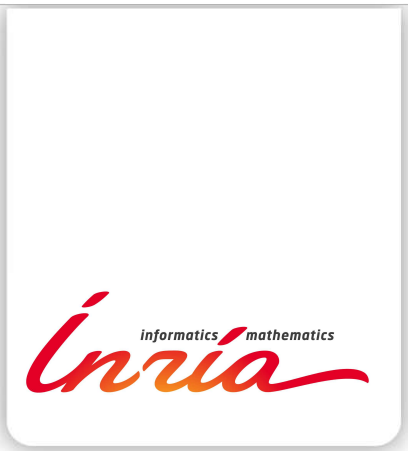
Proof. Similar to that of Proposition 9. □

Proposition 14:

Proof. Follows immediately from Relations 6 and 8 and $\tau_i^k > 0$. □

Proposition 15:

Proof. Follows immediately from Relations 6 and 8 and $\tau_i^k > 0$. □



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399